TURBO PASCAL

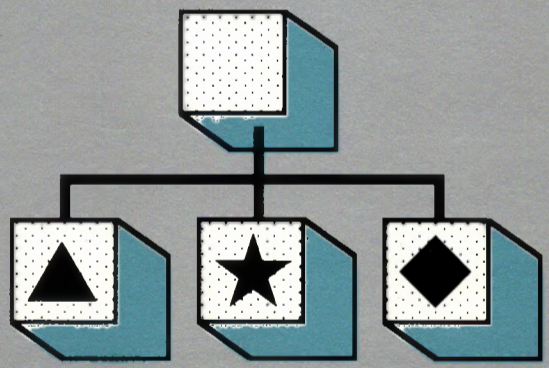
Volume 1

Screen Management

[redacted]

Object Professional™

A library of over 100 powerful object types containing over 1000 methods. Window object types let you use multiple overlapping resizeable windows providing: ■ Menus ■ Pick lists ■ Data entry ■ Text editing and much more. Data object types and system-oriented routines are included too!



TURBO Power

Object Professional

User's Manual

Volume 1
Screen Management

Copyright (c) 1989-1991, TurboPower Software
All Rights Reserved

Second Edition December 1991

P.O. Box 49009
Colorado Springs, CO 80949-9009

Technical Support: 719-260-6641
Fax: 719-260-7151
Sales: 800-333-4160
9 a.m. to 5 p.m. Mountain Time, Monday-Friday

CompuServe Mail: 76004,2611
CompuServe Support: PCVENB Section 6

Trademarks Mentioned

TurboPower Software and the distinctive TurboPower logo are trademarks of TurboPower Software.

Object Professional is a trademark of TurboPower Software.

Turbo Professional is a registered trademark of Sunny Hill Software, used under license to TurboPower Software.

Turbo Pascal, Turbo Assembler, Turbo Debugger, Turbo Database Toolbox, Turbo Editor Toolbox, SideKick, SideKickPlus, and SuperKey are trademarks or registered trademarks of Borland International.

DesqView is a trademark of Quarterdeck Office Systems.

Double DOS is a trademark of SoftLogic Solutions.

OmniView is a trademark of Sunny Hill Software.

Periscope is a trademark of The Periscope Company.

Hercules is a registered trademark of Hercules Computer Technology, Inc.

dBase is a trademark of Ashton-Tate.

Lotus is a trademark of Lotus Development Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

MS-DOS, OS/2, Windows, CodeView, and Macro Assembler are trademarks of Microsoft Corporation.

IBM PC, XT, AT, PS/2, and TopView are registered trademarks of International Business Machines Corporation.

Table of Contents

Volume 1: Screen Management

1. Introduction 1-1
   - A. System Requirements 1-4
   - B. Files Supplied 1-5
   - C. Installation 1-7
   - D. Organization of the Manual 1-14
   - E. Getting Help with POPHELP 1-17
   - F. DESKPOP: Desktop Manager 1-24
   - G. Purchase Agreement 1-45

2. Principles of Organization 2-1

3. Screen and Keyboard 3-1
   - A. Overview 3-1
   - B. OPCRT: Enhanced CRT Unit 3-2
   - C. OPCOLOR : Color Names 3-49
   - D. OPMOUSE: Mouse Routines 3-52
   - E. OPCMD: Command Processing 3-82

4. Window Objects 4-1
   - A. Overview 4-1
   - B. OPFRAME: Frame Objects 4-8
   - C. OPWINDOW: Window Objects 4-55
   - D. OPPICK: Pick Lists 4-186
   - E. OPDIR: Directory and Path Lists 4-241
   - F. OPMENU: Menu Systems 4-297

Index IN-1

Volume 2: Data Input/Output

4. Window Objects (continued) 4-361
   - G. OPHELP: Help Systems 4-361
   - H. Utility and Sample Programs 4-420

5. Text Editors and File Browsers 5-1
   - A. Overview 5-1
   - B. OPMEMO: Basic Text Editor 5-2
   - C. OPEDITOR: Advanced Text Editor 5-30
   - D. OPBROWSE: File Browser 5-56
   - C. Sample Programs 5-79

6. Data Entry 6-1
A. Overview 6-1
B. OPABSFLD: Abstract Field Definitions 6-4
C. OPFIELD: Data Entry Fields 6-15
D. OPFEDIT: Field Editors 6-52
E. OPSELECT: Field Selection 6-55
F. OPENTRY: Data Entry Screens 6-110
G. OPEDIT: Advanced Line Editor 6-176
H. OPSEDIT: Simple Line Editor 6-197
I. OPDATE: Date and Time Variables 6-206
J. Utility and Sample Programs 6-228

7. Printing 7-1
A. Overview 7-1
B. OPPRNLOW: Physical Level Printer Support 7-3
C. OPPRINT: Logical Level Printer Support 7-27
D. OPDEVICE: Hardware Specific Printer Support 7-57
E. OPFORM: Printed Forms 7-66
F. Sample Programs 7-113

Index IN-1

Volume 3: TSRs and More

8. Data Objects 8-1
A. Overview 8-1
B. OPROOT: Object-Oriented Data Structures 8-2
C. OPLARRAY: Object-Oriented Large Arrays 8-154

9. String Handling 9-1
A. Overview 9-1
B. OPSTRING: String Manipulation 9-2
C. OPASCIIZ: Long ASCIIZ Strings 9-35
D. OPSTRDEV: Formatted Strings 9-49

10. DOS/BIOS Calls 10-1
A. Overview 10-1
B. OPDOS: Assorted DOS/BIOS Calls 10-3
C. OPEXEC: Swapping Exec Manager 10-32
D. OPENHKBD: Enhanced Keyboard Interface 10-42

11. Interrupts and TSRs 11-1
A. Overview 11-1
B. OPINT: ISR Management 11-4
C. OPTSR: TSR Management 11-13
D. Overview of Swappable TSRs 11-34
E. OPSWAP1: Swappable TSRs, Part 1 11-47
F. OPSWAP: Swappable TSRs, Part 2 11-56
G. OP8087: 8087 Support 11-58
H. Sample Programs 11-64

12. Memory Allocation 12-1
A. Overview 12-1
B. OPEMS: EMS Memory Management 12-3
C. OPEXTMEM: Extended Memory Management 12-12

13. Macros 13-1
A. Overview 13-1
B. OPMACRO: Keyboard Macros 13-3
C. OPMACED: Macro Editor 13-15
D. OPREPLAY: Macro Playback 13-18
E. Sample Programs 13-23

14. Miscellaneous 14-1
A. Overview 14-1
B. OPBCD: High-precision Reals 14-2
C. OPCLONE: Program Customization 14-16
D. OPERRHAN: Runtime Error Recovery 14-25
E. OPINLINE: Assorted Inline Macros 14-28
F. OPSORT: In-memory Sort Routine 14-45

15. Appendices 15-1
A. What's New in Object Professional 1.0 15-1
B. Unit Dependencies 15-4
C. Object Hierarchy 15-7
D. Error Codes 15-10
E. Using Overlays with Object Professional 15-14
F. Converting Turbo Professional Applications 15-16
G. Compiling Large Programs 15-24
H. For Further Reading 15-26

Procedure and Function Reference PR-1

Index IN-1

Object Quick Reference

Volume 1: Screen Management

3. Screen and Keyboard
- ColorSet 3-48
- CommandProcessor 3-95
- CommandPacker 3-115

4. Window Objects
- ScreenRect 4-10
- AbstractFrame 4-31
- Frame 4-42
- RawWindow 4-62
- AbstractWindow 4-69
- StackWindow 4-118
- CommandWindow 4-143
- WindowStack 4-132
- LoadableColorSet 4-162
- PackedWindow 4-166
- VirtScreen 4-175
- PickList 4-208
- DirList 4-251
- PathList 4-285
- MenuItemNode 4-314
- Menu 4-323

Volume 2: Data Input/Output

4. Window Objects (continued)
- AbstractHelpWindow 4-384
- ScrollingHelpWindow 4-415

5. Text Editors and File Browsers
- Memo 5-10
- MemoFile 5-22
- TextEditor 5-41
- Browser 5-64

6. Data Entry
SelectField....6-18
EntryField....6-64
AbstractSelector....6-64
Selector....6-90
ScrollingSelector....6-103
EntryScreen....6-144
ScrollingEntryScreen....6-169
LineEditor....6-188
SimpleLineEditor....6-201

7. Printing
BasePrinter....7-10
DosPrinter....7-19
BiosPrinter....7-20
Printer....7-38
PrintField....7-88
PrintedForm....7-92
Report....7-109

Volume 3: TSRs and More

8. Data Objects
PointerStack....8-9
StaticQueue....8-21
SingleListNode....8-33
SingleList....8-34
DoubleListNode....8-43
DoubleList....8-44
CircularList....8-49
BitSet....8-52
LargeBitSet....8-64
StringDict....8-71
StringSet....8-80
StringArray....8-82
Root....8-99
IdStream....8-100
DosIdStream....8-120
BufIdStream....8-123
MemIdStream....8-125
DirEntry....8-132
Library....8-137
MemLibrary....8-148
AbstractArray....8-161

14. Miscellaneous
Cloner....14-19

1. Introduction

Object Professional is the latest generation of what we hope is considered to be the premiere library of subroutines for Turbo Pascal programmers. This succession of libraries has been around for almost five years now, and it has grown up much like a five year old child: by leaps and bounds, of course, but with the addition of new insights and maturity along the way.

Borland's introduction of Turbo Pascal 5.5 was--as with so many of their products--both a blessing and a curse for TurboPower Software. It was a blessing because it introduced object-oriented programming features to Pascal. Although none of us here had ever done any serious work with an OOP language, we quickly became OOP converts. This enthusiastic conversion lead to the curse, however: wherever it made sense, we decided to rewrite the Turbo Professional routines from scratch in object-oriented form. The coding, and especially the documentation, quickly swamped us with work.

We've finally reached the end of the project, however, and we think you'll like the result. You can feel the scope of the package just from the weight of the manuals and the size of the print, but here are some statistics: there are over 50 units, 130 object types, 2100 documented procedures, functions, and methods, 1200 additional internal routines, and over 1600 pages of documentation in **Object Professional**. Add to that a dozen commercial-quality demonstration programs and utilities, dozens of introductory example programs, and a comprehensive memory-resident help database. This is a big five year old!

In order to get a fast start with this wealth of information, we recommend one of two courses of action, depending on your level of experience. Veteran users of Turbo Professional should be sure to read Appendix A, "What's New in **Object Professional** 1.0," which gives a brief overview of the new features, followed by Appendix F, "Converting Turbo Professional Applications." Newcomers should continue reading this introduction, which will provide a brief overview of all the capabilities of **Object Professional**. Everyone should continue by reading section C of this chapter, "Installation," and section D, "Organization of the Manual." Chapter 2, "Principles of Organization," is also important; it provides an account of how the library is arranged and of the governing principles that shaped it. After that, browse through the table of contents and start dipping into specific topics that interest you.

The **Object Professional** units and demonstration programs fall into one of several broad categories:

Screen and Keyboard Handling
OPCRT is a plug-in replacement for the standard CRT unit, offering several extra capabilities and a host of enhancements. OPCOLOR simply defines a series of constants such as 'WhiteOnBlack' and 'BlackOnLtGray' that can be used to refer to particular video attributes. OPMOUSE offers a comprehensive set of low-level routines that simplify the task of adding mouse support to your programs. OPCMD provides facilities for translating keystrokes and mouse button presses into abstract "commands" such as "move cursor up," "select item," "quit," and so on.

Window Objects
OPFRAME defines several objects that work together to create windows with borders, shadows, titles, scroll bars, and mouse hot spots. OPWINDOW builds on top of OPFRAME to create a hierarchy of window objects ranging from simple ones to stackable, resizeable windows that are capable of interpreting keyboard input. OPWINDOW also provides objects for managing virtual screens and compressed screen images. OPPICK displays "pick lists," popup windows containing lists of items from which the user may choose. OPDIR combines with

OPPICK to create popup lists of filenames. OPMENU allows you to create sophisticated multi-level menu systems quickly and easily. OPHELP allows you to create context-sensitive help systems with a minimum of fuss. Three utility programs aid in using these units: PSCREEN is a memory-resident program that captures text screens and compresses them in a file; MAKEMENU lets you interactively design menu systems and then generates source code for them; and MAKEHELP compiles context-sensitive help text into compressed, indexed files for use by OPHELP.

Text Editor Objects
OPMEMO offers a simple free-form text editor that's perfect for note fields in business applications. OPEDITOR steps up to a powerful programmer's editor with block operations, find and replace, bookmarkers, and more. OPBROWSE uses virtual memory technology to display binary or text files of arbitrary size. Three useful demonstration programs--MEMO, EDITOR, and BROWSE--show how to use the units.

Data Entry
OPSEDIT provides a full-featured line editor to handle keyboard input. OPEDIT adds picture mask validation and numeric data entry. OPSELECT organizes groups of fields within a window, handling field-to-field movement but giving you control over what happens at each field. OPENTRY combines the powerful line editors used in OPEDIT with the window management of OPSELECT to provide an incredibly powerful data entry engine. OPDATE provides routines for working with date and time variables. MAKESCRN is a powerful utility program with which you can interactively design and generate source code for data entry screens. ENTRY and BEST example programs show how to combine the features of the data entry units into commercial quality applications.

Printing
OPPRNLOW routes output from your program to a printer and returns status information for error detection. OPPRINT adds the ability to manage print modes, typefaces, and control sequences. OPDEVICE has routines that allow OPPRINT to work on specific printers, Epson's and LaserJets especially. OPFORM is the printer's equivalent of OPENTRY, supporting the layout and printing of print fields, blocks, pages, forms, and reports. The FORMTEST example program demonstrates how to put all of these concepts into action.

Data Objects
OPROOT defines a number of general-purpose objects that are often referred to by OOP afficionados as "container classes"--stacks, queues, lists, sets, and so on. Many of the **Object Professional** units use and extend these objects to provide higher level capabilities; you can too. OPLARRAY lets you break through the 64K array size barrier in object-oriented style, storing the raw data in RAM, in EMS memory, or on disk.

String Handling
OPSTRING contains a comprehensive set of basic routines for manipulating standard Turbo strings. OPASCIIZ is a similar unit that allows you to work easily with strings larger than 255 characters. OPSTRDEV lets you use standard Write statements to create formatted strings.

DOS/BIOS Interface
OPDOS gives you easy access to a host of DOS and BIOS services not readily available using the standard DOS unit. OPEXEC provides a swapping DOS shell which lets you squeeze two programs temporarily into the space of one. OPENHKBD simplifies the task of writing programs that support the enhanced keyboard.

Interrupts and TSRs
OPINT manages your interrupt service routines (ISRs) for you. OPTSR contains everything you need to write a commercial quality TSR with remarkably little effort. Even more remarkable, OPSWAP and OPSWAP1 form a system for writing swappable TSRs that consume as little as 6K of DOS RAM. OP8087 contains low-level routines needed when writing TSRs that use the 80x87 math chip or Turbo Pascal's floating point emulator, as well as routines that allow you to recover from otherwise fatal arithmetic errors. The PTIME utility program demonstrates how to combine real-time interrupt service routines with a hotkey-activated TSR. PSCREEN, already mentioned once, is a relatively simple example of a swappable TSR. DESKPOP is a very full-featured swappable TSR that stuffs a complete programmer's desktop manager into only 6K of DOS RAM.

Memory Allocation
OPEMS gives you easy access to expanded memory (EMS), while OPEXTMEM unlocks the door to the protected mode on '286 and '386 machines, letting you use extended memory for data storage.

Keyboard Macros
OPMACRO lets you easily add a keyboard macro facility to your programs, and OPMACED contains a sophisticated macro editor that goes with it. OPREPLAY is a streamlined version of OPMACRO that is especially useful for implementing a "paste" function in a swappable TSR.

Miscellaneous Routines
OPBCD provides a comprehensive set of routines for doing high-precision arithmetic using BCD (Binary Coded Decimal) reals. OPCLONE assists in the writing of self-modifying programs and installation programs. OPERRHAN provides a facility that allows you to recover from most runtime errors. OPINLINE contains a collection of handy inline macros, as well as some routines for performing pointer arithmetic. OPSORT gives you a speedy, non-recursive quicksort that works with almost any data type. DPINST, the installation program for DESKPOP, shows how to use OPCLONE.

A. System Requirements

To use this package, you must have the following:

1. An IBM PC, XT, AT (or close compatible), or PS/2 running DOS 2.0 or later.

2. Turbo Pascal version 5.5 or 6.0.

3. At least 512K of RAM. 640K of memory is strongly recommended. If you will use only some of the **Object Professional** units, you may be able to survive with much less memory. If you will be debugging large programs using Turbo Debugger, 1-2 megabytes of EMS memory is also recommended.

4. Two floppy disk drives. A hard disk is strongly recommended. To install all **Object Professional** files (TPU, OBJ, EXE, PAS, INC, ASM) requires about 11 megabytes of free disk space. To retain just the TPU files, only a little over a megabyte is required. If you've purchased the package for just the TSR manager (or perhaps some other isolated feature) you could work on a floppy-based system.

5. An 80-column monitor.

Optional:

6. Borland's Turbo Assembler (TASM) 1.0 or higher, or Microsoft's Macro Assembler (MASM) 4.0 or higher. Needed only if you wish to modify any of the assembly language routines.

7. A Microsoft-compatible mouse. All of the user-interface capabilities of **Object Professional** support the mouse.

B. Files Supplied

Object Professional is distributed on ten 360K 5.25" or five 720K 3.5" diskettes. One disk contains the INSTALL program and some demonstration programs. Five contain the source code for library units and demonstration programs. Two diskettes contain pre-compiled executable files for the demonstration programs. One diskette contains a memory-resident help system and the help files. The last diskette, a "BONUS" disk, contains various freely distributable units and programs.

The majority of the files on the distribution diskettes are compressed with the freely distributable utility LHARC. The INSTALL program will automatically decompress all files as it copies them. We've also provided a copy of the utility LHARC so that you can decompress files outside of the INSTALL program.

The following table outlines the files that you'll find on the distribution diskettes.

INSTALL/DEMO Disk:

READ.ME
If you see a file of this name on the diskette, please read it before proceeding. It will contain the latest updates to the printed documentation.

README.LZH
If you find this file, it contains a compressed text file (READ.1ST) that provides additional information we couldn't fit into READ.ME. INSTALL will decompress it automatically or you can do so manually by copying it to another disk and entering LHARC X README.

FASTUPD.LZH
A compressed file (FASTUPD.nnn) containing detailed information about changes specific to this version of **Object Professional.**

INSTALL.EXE
Installation program for **Object Professional** 1.0.

OPRO.MAK
This file is to be used in conjunction with Borland's MAKE utility (provided with Turbo Pascal). It provides instructions telling MAKE how to rebuild all the **Object Professional** units and many of the demo programs.

LHARC.EXE & LHARC.MAN
A utility for decompressing LZH files. You'll need this if you want to use the manual installation method, described below. You'll also find a compressed manual for LHARC on the BONUS disk. LHARC.MAN is the documentation.

PRINTDOC.EXE
A utility for printing text files. Useful for printing the READ.ME or READ.1ST files with pagination and margins. Type PRINTDOC alone to get a list of options.

SRC1 Disk:

OPPAS1.LZH contains Turbo Pascal source code for the library routines.

SRC2 Disk:

OPPAS2.LZH contains more Turbo Pascal source code for the library routines.

SRC3 Disk:

OPASM.LZH contains the source code and OPOBJ.LZH the object files for all routines written in assembly language. DEMOSRC1.LZH contains the Turbo Pascal and assembly language source as well as the object code for about half of the demonstration programs.

SRC4 Disk:

OPPAS3.LZH contains more Turbo Pascal source code for the library routines. OPDIALOG.LZH contains Turbo Pascal source code and documentation for the SAA/CUA-style dialog box units. OPDRAG.LZH contains Turbo Pascal source code and documentation for the mouse-dragging unit. OPXMS.LZH contains Turbo Pascal source code and documentation for the XMS (extended memory) access unit.

SRC5 Disk:

DEMOSRC2.LZH contains the Turbo Pascal and assembly language source as well as the object code for about half of the demonstration programs. EXAMPLES.LZH contains source code for the introductory example programs. The listing of each program, printed at the appropriate location in the manual, shows the associated file name.

EXE1 Disk:

EXE1.LZH contains several executable demonstration programs.

EXE2 Disk:

EXE2.LZH contains several more executable demonstration programs.

HELP Disk:

POPHELP.LZH contains executable and source files to the POPHELP memory-resident help system. It also contains the help text files.

BONUS Disk:

READ.ME! contains brief descriptions of other LZH files found on this disk. The other *.LZH files are various programs and units that have been released for free distribution. Several of them were written by TurboPower Software, others are included with the permission of the author.

C. Installation

In this section we'll show how to transfer **Object Professional** from its highly compressed form on the distribution diskettes to usable files on your working drive. As we mentioned earlier, a hard disk is highly recommended for using **Object Professional**. This installation guide focuses on the methods for a hard disk; if you are attempting to use a floppy-based system, you should call us for advice if the interactive method doesn't work.

The library units for **Object Professional** 1.0 are distributed in source form only. In order to use any of the units, you'll need to compile them first. We've provided two different methods for you to do this.

The first method, the easier to use, involves an interactive program called INSTALL.EXE, which you'll find on the INSTALL/DEMO diskette. With this program, you can select any number of the **Object Professional** units, and INSTALL will automatically copy, dearchive and compile them to locations of your choice. INSTALL is especially useful if you're cramped for disk space, if you don't want to install all of the units, or if you're uncomfortable with Unix-style make files. **Object Professional**'s INSTALL also has a nice new feature (compared to Turbo Professional's): you can copy the archive files into a directory on your hard disk and INSTALL will access that directory rather than the floppies. In this way, you can store your master diskettes, let INSTALL create the TPU files you need, delete the source files you're not using at the moment, and rerun INSTALL to grab a few source or TPU files when you need them.

The second method, more powerful but less friendly, counts on Borland's MAKE.EXE utility and the supplied make file OPRO.MAK. The make file contains a complete set of dependencies for all of the **Object Professional** units and many of the demonstration programs. This method is somewhat faster than running INSTALL if you want all of the **Object Professional** files immediately available. The make file is also capable of reassembling OBJ files when the ASM source has been modified. (Note that we supply pre-assembled OBJ files.) You must have at least 11 megabytes of free space on your disk to use the make file method.

No matter which method you choose, we recommend that you use three different subdirectories to store the **Object Professional** files:

```
\OPRO       for Pascal source and TPU files
\OPROOBJ    for assembly language source and object files
\OPRODEMO   for demonstration/utility source and executable files
```

If all of the files are kept in a single directory, DOS performance when opening files (during compilation) is severely degraded. Both the INSTALL program and the make file have provisions for supplying your own names for these three directories.

Make File Installation Method

To use the make file approach, choose or create the destination subdirectories as described above. You'll want these directories on the Turbo compiler's search paths for source, OBJ and TPU files. You may also want to put this directory on your DOS path, in order to run the standalone utilities such as SMACS and MAKEMENU. Alternatively, once you've installed **Object Professional**, you can move the EXE files to another directory that's already on the DOS path.

Remember that you need at least 11 megabytes of free disk space to use the make file installation method. (Once installation is complete, you can erase many of the files, reducing the space requirement to just over 1 megabyte if you keep just the TPU files around.) If you have less space, skip ahead to the INSTALL program method.

To continue with the make file installation, copy all files and archives from each **Object Professional** diskette to the proper directory on your hard disk, as shown in the following table:

Diskette/File	Destination
INSTALL/DEMO Disk:	
READ.ME	\OPRO
README.LZH	"
FASTUPD.LZH	"
INSTALL.EXE	"
OPRO.MAK	"
LHARC.EXE	utility directory on dos path
LHARC.MAN	"
PRINTDOC.EXE	"
SRC1 Disk:	
OPPAS1.LZH	\OPRO
SRC2 Disk:	
OPPAS2.LZH	\OPRO
SRC3 Disk:	
OPASM.LZH	\OPROOBJ
OPOBJ.LZH	"
DEMOSRC1.LZH	\OPRODEMO
SRC4 Disk:	
OPPAS3.LZH	\OPRO
OPDIALOG.LZH	"
OPDRAG.LZH	"
OPXMS.LZH	"
SRC5 Disk:	
DEMOSRC2.LZH	\OPRODEMO
EXAMPLES.LZH	"
EXE1 Disk:	
EXE1.LZH	\OPRODEMO
EXE2 Disk:	
EXE2.LZH	\OPRODEMO
HELP Disk:	
POPHELP.LZH	\OPRODEMO
BONUS Disk:	
all files	separate directory of your choice

As noted, you should copy LHARC (and its documentation) to a utility directory somewhere on your DOS PATH. We'll take advantage of that in the following.

Once you've copied the files into the specified locations, go to each directory, decompress the files, and delete the archives:

```
CD\OPRO
LHARC X *.LZH
DEL *.LZH
CD\OPROOBJ
LHARC X *.LZH
DEL *.LZH
CD\OPRODEMO
LHARC X *.LZH
DEL *.LZH
```

Actually, if you're a little tight on disk space but have time to spare, you can decompress the files directly from the master diskettes. For example, to decompress the OPPAS1.LZH file from the SRC1 diskette, you would put SRC1 into your A: drive, make your working drive current, and execute the following sequence from the DOS prompt:

```
CD\OPRO
LHARC X A:OPPAS1
```

The LZH files contain no duplicate files. If LHARC detects that it is about to overwrite an existing file (because your directory wasn't empty to begin with), it will prompt for confirmation before proceeding. If needed, press <CtrlC> to stop LHARC and return to the DOS prompt.

Before proceeding, you may need to modify OPRO.MAK to work with your drive and directory setup. The make file specifies such information as the path to the Turbo Pascal command line compiler and whether you have an assembler or not. You may need to modify the following macros and defines:

```
asmdir        if you've chosen a directory besides \OPROOBJ
tpudir        if you want TPU files in other than \OPRO
demodir       if you've chosen a directory besides \OPRODEMO

adir          same as asmdir, but ending in backslash
tdir          same as tpudir, but ending in backslash
ddir          same as demodir, but ending in backslash

HaveAssembler if you have MASM or TASM and want to be able to reassemble
MakeDemos     if you want the make file to recompile demo programs as well as
              TPU and OBJ files
tasm          if you HaveAssembler but have MASM instead of TASM

assemble      to specify the path to the assembler
compile       to specify the path to TPC.EXE and any special compiler options
```

The make file itself contains detailed instructions, just use your favorite text editor. If you're not comfortable making minor edits to a make file, you should use the INSTALL.EXE installation method.

Next, compile everything using Borland's MAKE utility:

```
MAKE -fOPRO.MAK
```

Don't try to use Microsoft's MAKE (or any other MAKE utility) with the supplied make file. It is sufficiently different from Borland's that it just won't work. Note that the "-f" command line option must be entered in lowercase just as shown.

With the correct settings in OPRO.MAK, MAKE will build the entire library without errors. If you obtain any errors, double check the make file. If this doesn't help, check for a TPC.CFG file. (See the Turbo Pascal manual for a description of TPC.CFG.) If you find a TPC.CFG file, erase it or temporarily rename it while MAKE runs.

You can optionally build the POPHELP (memory resident help utility) database for **Object Professional** at this time. To do so, enter the following commands:

```
CD\OPROOBJ
MAKEHELP /Q OPRO
```

MAKEHELP reads the help text files and creates a file named OPRO.HLP.

If you're short on disk space, you can now erase a number of the files (at least until the next time you need to modify or rebuild one of our units). To remove all but the TPU and EXE files, enter the following DOS commands:

```
CD\OPRO
DEL *.PAS
```

```
DEL *.I*
CD\OPROOBJ
DEL *.ASM
DEL *.OBJ
CD\OPRODEMO
DEL *.PAS
DEL *.I*
DEL *.ASM
DEL *.OBJ
DEL *.TPU
DEL *.TX*  (only if you've already run MAKEHELP to create OPRO.HLP)
```

This leaves only the all-important TPU files and the EXE files for the demonstration and utility programs.

If you haven't already done so, take this opportunity to print and read the READ.ME, READ.1ST, and FASTUPD.nnn files, which provide information to update this manual.

INSTALL.EXE Installation Method

INSTALL is an interactive tool we've written to make it very easy for you to dearchive, copy, and compile the **Object Professional** units. With it, you can select only as many **Object Professional** files as you actually need. INSTALL even figures out the unit dependencies for you, so you get every file needed to compile a given program.

You can trade off disk space for time when you run INSTALL. If you have an extra couple of megabytes of disk space, INSTALL will run much faster if you first copy all of the master diskettes into a single archive directory. In the long run, this may save you space since you can have INSTALL erase all but the TPU and EXE files, and conveniently rerun INSTALL to get selected source files if and when you need them. To take advantage of this option, create a directory (the name is arbitrary) on your hard disk and copy all files from the **Object Professional** diskettes to that directory.

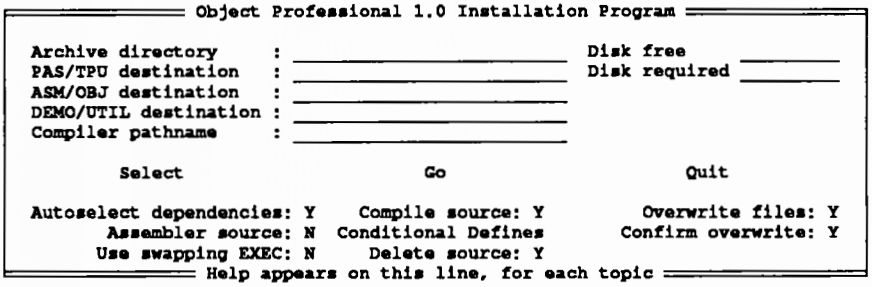
If you're installing directly from the floppy diskettes, insert the INSTALL/DEMO disk in drive A and enter

```
A:INSTALL
```

If you're using the archive directory method, run the copy of INSTALL in that directory.

When it starts up, INSTALL assumes you'll be installing from the drive where INSTALL is located to the current drive and directory. It also searches the DOS path for your copy of TPC.EXE, which it will use to compile the **Object Professional** units.

When INSTALL's main window appears, you'll see a screen something like the following:



The first four lines show the default directories. Archive directory refers to the location of the compressed LZH files. This will typically be either A:\ or a special archive directory that you've created and loaded with the contents of the distribution diskettes. The next three lines specify the destination directories for the decompressed files. The fifth line specifies the location of TPC.EXE, Borland's Turbo Pascal command line compiler.

If you want to change INSTALL's default assignments or if INSTALL wasn't able to find TPC.EXE, use the cursor keys to position the highlight bar over the directory to change, press <Enter> to select it, type in the desired string, and press <Enter> once more to accept the changes. You can also press <Esc> to undo any changes you made while editing an item.

The archive and destination directories require a drive and a directory name. The archive directory (only) can contain a list of alternate directories separated by semicolons (like the DOS PATH command). If you change the PAS/TPU destination directory, INSTALL will automatically update the ASM/OBJ and DEMO/UTIL directories to the PAS/TPU name plus the letters 'OBJ' and 'DEMO', respectively. Note that INSTALL will create the specified destination directories if they do not already exist.

The compiler pathname can be either a drive and directory or a complete path and filename (in case you've changed the name of your TPC.EXE). The specified compiler MUST be a version 5.5 or later compiler or you'll get errors during compilation.

If you're installing on a two-floppy system, you'll probably want to enter B:\ for all destination directories, and A:\ for TPC.EXE directory. In that case, you'll be swapping **Object Professional** distribution diskettes into drive A:, and putting fresh output diskettes into drive B:. When the time comes to compile, INSTALL will prompt you to insert a diskette containing TPC.EXE and TURBO.TPL in drive A:. When limited RAM space is available, you may also need a diskette containing COMMAND.COM, in case DOS must reload this file after a compile.

In the upper right-hand corner of the window is the disk space display. "Disk free" is the approximate number of bytes currently available on the PAS/TPU destination drive. This number will be updated as INSTALL copies and compiles files onto that drive. "Disk required" shows the minimum number of bytes required to write the currently selected **Object Professional** files to the destination drive. Of course, you'll want to assure that free space exceeds required space. INSTALL doesn't enforce this requirement, leaving it to you to select a group of files that fit into the available space. Note that INSTALL's calculation of required space is not exact, although it is a close estimate.

The center row, appearing in a different color on your screen, shows INSTALL's "action buttons." When you position the highlight bar over any of these items and press <Enter>, INSTALL takes further action.

The Select action causes INSTALL to display a second window, the File Selection Window. In this window, you can select the **Object Professional** units and demo programs that you'd like to install. Press the cursor keys to position the highlight bar, the <Ins> key to select a particular unit, the <Del> key to deselect a unit, <Enter> to toggle the selection, or <CtrlEnter> to clear all current selections. Selected units are displayed with an arrow character on the left. You'll note that the "Disk required" indicator is updated after each selection.

When you select any particular unit or demo program, INSTALL is actually selecting a whole group of files that are required to compile that unit (Pascal files, include files, OBJ files, and so on). If the "Autoselect dependencies" option is On (more on this below), INSTALL selects any units on which the current program or unit depends. It's actually entertaining to see the units

pulled in by each program. You can experiment by alternately selecting and pressing <CtrlEnter>. Note that automatic dependency selection works only in one direction, however. When you deselect a unit, the units on which it depends are not deselected with it.

When output is being written to a floppy, the complete results of each set of file selections must fit onto one floppy. This requirement will severely limit INSTALL's ability to work on a floppy disk system.

This is a good place to reiterate that you should always select the READ.ME file for copying. This file contains information that didn't make it into the printed manual, and it's often very important stuff, indeed. We can't begin to count the number of embarassed customers who have spent money on long-distance support calls only to learn that the answers to their questions were already in the READ.ME file.

Once you've selected the set of files you want, press <Esc> and the Selection window will close, returning you to INSTALL's Command window.

The other two action buttons, Go and Quit, do what their names suggest. Go starts the process of dearchiving, copying and compiling. Quit returns you to the DOS prompt. Before you actually proceed to the Go action, read on while we describe the rest of INSTALL's options.

The bottom three rows of the Command window are options controlling how INSTALL behaves. Position the highlight bar over any of the Y/N items, and press <Enter> to toggle the setting.

We've already mentioned "Autoselect dependencies." When this is set to Y, the Selection window automatically activates any units on which a chosen unit depends. Because **Object Professional** has so many files, you may experience a noticeable delay when this option is enabled and you select multiple files.

When "Assembler source" is set to Y, INSTALL dearchives and copies assembly language source files to the destination. Otherwise, it copies only the pre-assembled OBJ files. You won't need the assembly source unless you want to study or modify it.

When "Use swapping EXEC" is Y, INSTALL swaps the majority of its code to EMS (or disk) before invoking the compiler. This gives the compiler more working memory. Don't select this option if you're running on a floppy-based system without at least 150K of EMS memory free.

When "Compile source" is Y, INSTALL runs the Turbo Pascal compiler (TPC.EXE) to compile each unit and program you've selected, thus creating TPU and EXE files for further use.

The "Conditional defines" option brings up a list of conditionally defined symbols. Use the cursor keys to select (or deselect) desired options. The OPDEFINE.INC file will be updated (at compile time) with your changes. See the next chapter, "Principles of Organization," for more information about each define. Note that certain **Object Professional** demonstration programs (not units) require specific defines. We are providing precompiled EXE files for the most critical utility programs so that this issue won't affect you in any significant way. However, if you disable a required define, TPC will generate an error while compiling the demonstration program. You can continue to compile other programs by simply pressing <Enter> to clear INSTALL's error window.

Set "Delete source" to Y if you don't want to leave the source files on your disk. INSTALL will then delete all PAS, OBJ, ASM and include files, leaving only the TPUs and the executable demonstration programs.

When "Overwrite files" is Y, INSTALL will overwrite any file of the same name in the destination directory, subject to your confirmation if the "Confirm overwrite" setting is also Y. By setting "Overwrite files" to N, you can easily update your **Object Professional** directory with additional files from the distribution disks without wasting time on files that already exist.

After you've set these options to your liking, and selected the files you want, you're ready to Go. INSTALL will prompt you to load the appropriate **Object Professional** diskettes. You'll be able to watch the progress of installation in the DOS window on the lower portion of the screen. If anything goes wrong, INSTALL will prompt whether you want to continue or quit the current set of actions. When INSTALL finishes successfully, press <Esc> to clear the status message, then select the Quit action to return to DOS, or select another group of files.

D. Organization of the Manual

For a library the size of **Object Professional**, organization of the information is critical. We can't expect everyone to read and memorize 1600 pages of documentation–although we hope you do!–so the next best thing is to make the documentation easy to use now and in the future. We've attempted to provide everything that you need to find answers to your questions quickly. This section will describe how you can use these manuals most effectively.

As you know, the manual is divided into three volumes. This was originally driven by production issues (wire-bound books can be made only so thick), but the division should also help you to find information faster. The manual's binding style was chosen specifically so you could pick the right volume off the shelf the first time. Each volume contains topics that are logically related:

```
Volume 1: Screen Management
Volume 2: Data Input/Output
Volume 3: TSRs and More
```

Limitations on page count have forced us to make a few compromises on placing units in the volumes. Specifically, the OPHELP unit and the discussion of MAKEMENU have been pushed from Volume 1 where they most naturally fit into Volume 2. Also the discussion of data objects--OPROOT and OPLARRAY--was pushed out of Volume 2 into Volume 3, which is kind of a catch-all collection of topics in any case.

Each book begins with a complete table of contents and ends with a complete topic index. Frequent users of the manual will probably find the table of contents to be the quickest path to finding a particular routine: we try to limit the degree of detail in the table of contents so that you can scan it quickly.

The final volume ends with a procedure/function/method index. This index is basically a listing of the documented portions of the interface sections of all **Object Professional** units, in declaration order, annotated with a volume, chapter, and page reference for each routine. If you can remember that you need a routine from a particular object or unit, but you can't remember its name or parameters, this index will be invaluable to you.

Chapters are divided along topical boundaries. A chapter is then subdivided into sections, each describing a unit. A section is typically organized something like this:

```
4. Window Objects                                    (Topical Chapter)
  E. OPDIR - Directory and Path Lists                 (Unit Section)
    Unit Overview
    1. DirList: Filename Selection                    (Object Subsection)
      Object introduction
      Introductory example program
      Special topics
      Declarations
        Constants
        Types
        Variables
      Methods
    2. PathList: Path Selection
      ...
    3. Procedures and Functions
      ...
```

In the Types section, you will often see an entry like this:

```
RawWindow =
  object(AbstractWindow)
    ...
    wFrame          : Frame;
    aFrame          : Frame;
    ...
  end;
```

The ellipses ("...") indicate that the object declares one or more data fields that are not documented in the manual. If there are no ellipses, the object either has no data fields, or has only those inherited from its parent.

To make it easier to find the discussion of a given procedure or method, the names of the routine(s) being discussed on a given page are shown at the top of the page, followed by a solid line. The bottom of the page shows the name of the unit being documented and, if appropriate, the object. So in the case of the CommandWindow.Process method, "Process" would appear in bold type at the top of the page along the outer edge. The bottom of the page would show the unit name (OPWINDOW) and page number on the outside edge, and the object name (CommandWindow) on the inside edge.

Some sections begin with a diagram showing the hierarchy of objects to be discussed:

```
[CommandWindow]
  [AbstractSelector]
    [Selector]
      [ScrollingSelector]
    EntryScreen
      ScrollingEntryScreen
```

The names in brackets indicate members of the hierarchy that are documented elsewhere.

If you have time to read any of the documentation in advance, we recommend that you read the unit overview, object introduction, introductory example program, and special topics for each section. We've put a lot of effort into preparing complete example programs for each object or unit, knowing that many people learn best by example. These introductions tie everything together; the Declarations and Methods sections follow up with reference detail.

Each introductory example program is also provided in source form on disk so you don't have to type it in. The name of the source file is in a comment to the right of the program name.

You'll note that non-object-oriented procedures and functions are listed separately from the methods. Most units are slanted heavily one way or the other, so the distinction isn't important often, but it's worth keeping in mind.

Each chapter ends with a section that describes the utility and sample programs built using the routines just described. Some of these programs, MAKEMENU and MAKESCRN especially, are essential reading since they'll generate much of your source code for you. All of the sample programs have important lessons to teach, so don't overlook them.

Methods, procedures, and functions are documented one to a page wherever possible. Due to space and cost limitations we are forced to merge more than one routine per page where they will fit neatly. Even so, the names of all routines on the page are printed at the top of the page in bold type. Since the routines are listed in alphabetical order, you can flip through the manual to find a specific routine fairly quickly. The unit name and, where appropriate, object name are shown on the last line of each page so that you can flip to find those as well.

Within each routine's description, the following categories are shown:

Declaration
The routine's Pascal declaration showing a complete parameter list

Purpose
A one-sentence description of the routine's purpose

Description
More details about the parameters, operation, side-effects, and possible error conditions of the routine

Examples
Code fragments or short programs showing how to use the routine, with commentary

See Also
Related routines. If the related routine is in the same object or unit, it is listed just by name. If it is a method of another object, it is shown as ObjectName.MethodName. If it is a non-object-oriented routine in another unit, it is shown as RoutineName (UNITNAME).

Don't forget to take a look at the Appendices. These include some very useful tables–unit dependencies, the object hierarchy, error codes, overlayable units–as well as advice about converting Turbo Professional applications to **Object Professional** and compiling large programs with Turbo Pascal.

Also please remember to read the READ.ME file. It's a good idea to print it out, so that a few months from now you'll find it sticking out from the edge of your manual and remember to look at it again. The READ.ME file answers common questions, describes new features of the software, corrects errors in the manual–all very important stuff to know.

Finally, we've provided a computer-readable subset of the documentation and a powerful utility to access it. Read on to learn about POPHELP, which may be the best way to keep your **Object Professional** manuals from wearing down to a frazzle.

E. Getting Help with POPHELP

POPHELP is a memory-resident program designed primarily for providing easy-to-access, online help for the **Object Professional** library. Since it is based on the advanced TSR swapping capabilities of the OPSWAP unit, it can occupy as little as 6K of DOS memory. POPHELP uses help files generated by **Object Professional**'s MAKEHELP utility, so you're also able to merge help files from different TurboPower products or to extend the help on your own.

POPHELP is a new version of the GUIDE program that we've distributed with recent versions of Turbo Professional. The help files that POPHELP reads are not compatible with GUIDE's. However, you can recompile older help text files using the MAKEHELP utility supplied with **Object Professional** and bring them up to date with POPHELP standards.

Installing POPHELP

POPHELP can be installed in several ways:

- as a normal TSR using 250K of RAM
- as a TSR that swaps itself to EMS (expanded) memory, leaving only an 8K kernel behind in normal RAM
- as a TSR that swaps itself to disk, leaving an 8K kernel in RAM
- as a shell that execs another program and then automatically removes itself from memory.

You select among these options by using POPHELP command line parameters.

(You may wonder about the discrepancy between the 6K and 8K RAM usage numbers. The extra 2K is used by POPHELP's block paste facility, which allows you to paste portions of the help text into your editor. If you'd like to cut RAM usage to 6K, you can change a POPHELP conditional define and recompile it. The technique for doing so is described in POPHELP.PAS.)

Supplied with this version of POPHELP is a help file for **Object Professional** 1.0. This help file contains information about all interfaced objects, methods, constants, types, variables, procedures, and functions in the library. In addition, it provides useful tables of information about scan codes and line drawing characters. The help file is supplied in source format so that you can add to it as you see fit. Compiling it is a simple one-step operation.

If you selected POPHELP with the INSTALL program, it put the following files in the \OPRODEMO directory:

```
POPHELP.EXE
POPHELP.PAS
PHMAIN.PAS
PHTIMER.ASM
PHTIMER.TPU
OPRO.TXT
OPRO.T*       (help text include files)
OPRO.HLP
MAKEHELP.EXE
```

If you didn't run INSTALL, you'll need to compile the help text to create OPRO.HLP. Do so by typing:

```
MAKEHELP /Q OPRO
```

Once you've created the OPRO.HLP file, you can delete the OPRO.T* files from your disk. If you're not interested in studying or modifying POPHELP, you can also delete the source files from your disk. This should leave you with only the two required run-time files for POPHELP:

```
POPHELP.EXE      the popup reference engine
OPRO.HLP         the Object Professional help file
```

These two files consume about 900K of disk space.

POPHELP is set up to load the OPRO.HLP file by default. Hence, in the simplest case, you make POPHELP memory resident by entering

```
POPHELP
```

at the DOS command line while OPRO.HLP is in the current directory. This will install POPHELP in swapping mode; it will use EMS memory (about 250K) or disk space (two files, each about 250K) for swapping and will retain about 8K of your DOS RAM.

POPHELP provides a number of options to control how it goes resident. It uses the following DOS command line syntax:

```
POPHELP [HelpFile] [Options]
```

HelpFile is an optional pathname to a help file. POPHELP will automatically load the OPRO.HLP file by searching in the current directory, in the directory of POPHELP.EXE (if you're running DOS 3.0 or later), and on the DOS PATH. Specify a different directory or a different help file by putting it on the POPHELP command line. POPHELP applies a default extension of HLP to the specified file name.

The following options may be specified on the command line:

```
/B          Force use of black and white video attributes.
/D          Force swapping to disk even if EMS is available.
/E ProgToExec CommandLine
            Execute a program instead of going resident. No swapping. This
            option must be the last one on the command line.
/H Height   Specify initial help window height other than 12 rows.
/I HotKey   Specify alternate hot key for help index.
/L HotKey   Specify alternate hot key for screen lookup.
/M          Disable message appearing while TSR swaps to or from disk.
/N          Force entire TSR to remain in memory (no swapping).
/P HotKey   Specify alternate hot key for previous topic.
/S Path     Specify drive and directory for disk swapping.
/U          Unload TSR from memory.
/?          Display a help screen.
```

Examples:

```
POPHELP C:\OPRODEMO\OPRO
```

Load OPRO.HLP from the specified directory.

```
POPHELP MYHELP /H 15
```

Load MYHELP.HLP using a default help window height of 15 rows.

```
POPHELP /D /M /S F:\
```

Force POPHELP to swap to disk, using the root directory of drive F: for the swap files. Disable swap messages.

```
POPHELP /E TURBO MYPROG.PAS
```

Load POPHELP temporarily and run the TURBO integrated environment.

```
POPHELP /N
```

Make POPHELP resident without swapping.

```
POPHELP /U
```

Unload the resident copy of POPHELP.

```
POPHELP /B
```

Force POPHELP to use only black and white ($07, $0F, $70) video attributes.

When the /E option is used, POPHELP is resident only for the time the specified program is running. When the program halts, POPHELP removes itself from memory automatically. POPHELP uses COMMAND.COM to run ProgToExec, so the program can be anywhere on the DOS PATH. You can also specify any command line parameters normally used by the program, with the exception that redirection of input or output is not allowed.

If you have extended memory, the best way to let POPHELP take advantage of it is with a RAM disk. Use the /S option to give POPHELP the drive letter of the RAM disk, and it will swap to and from that area. For a help file the size of OPRO.HLP, you'll need about 360K free space on the RAM disk.

While POPHELP swaps to and from disk, it normally puts a status message on the top line of the screen. If you're swapping to a RAM disk, this status line may be an annoyance. If so, specify the /M option to disable the status.

Since the help window is resizeable after POPHELP pops up, the /H option is of minor use. It is supported because GUIDE (which didn't allow resizing) had it.

POPHELP uses three different hot keys. By default, the hot keys are:

<LeftShift> <F1>
Look up help topic based on word at cursor

<LeftShift> <F2>
Redisplay previous help topic

<LeftShift> <F3>
Display help index

The following options control the hot keys:

```
/L    Look up help topic
/P    Redisplay previous help topic
/I    Display help index
```

You can specify alternate hot keys at the time POPHELP is installed. On the command line, a hot key is specified as a hexadecimal word. The top byte specifies the shift key(s) to be pressed and must be non-zero. The bottom byte specifies the scan code for the hot key.

The shift key codes are:

```
RightShift - 01
LeftShift  - 02
Ctrl       - 04
Alt        - 08
```

Valid scan codes (in hexadecimal) are:

```
A - 1E    N - 31    0 - 0B    F1 - 3B    [ - 1A
B - 30    O - 18    1 - 02    F2 - 3C    ; - 27
C - 2E    P - 19    2 - 03    F3 - 3D    , - 33
D - 20    Q - 10    3 - 04    F4 - 3E    / - 35
E - 12    R - 13    4 - 05    F5 - 3F    \ - 2B
F - 21    S - 1F    5 - 06    F6 - 40    ] - 1B
G - 22    T - 14    6 - 07    F7 - 41    ' - 28
H - 23    U - 16    7 - 08    F8 - 42    . - 34
I - 17    V - 2F    8 - 09    F9 - 43    ` - 29
J - 24    W - 11    9 - 0A    F10- 44
K - 25    X - 2D              F11- 57
L - 26    Y - 15              F12- 58
M - 32    Z - 2C
```

For example:

```
/L 0244    performs lookup when <LeftShift><F10> is pressed
/P 0819    shows previous topic when <Alt><P> is pressed
/I 0517    displays help index when <Ctrl><RightShift><I> is pressed
```

Two hot keys may not be based on the same scan code, even if the shift keys differ. For example, you cannot use <Alt><F1> for one hot key and <Ctrl><F1> for another. You must specify at least one of the shift keys for each hot key, i.e., the top byte of the specified word must not be zero.

Keep in mind one important restriction when installing POPHELP in swapping mode. When POPHELP pops up, it disables any TSRs loaded after it. This is necessary because it overwrites their memory while it swaps in. Some TSRs, especially network shells and communication programs, won't accept being disabled in this way: the network will disconnect a station that doesn't respond after a time, or a communication program will lose incoming characters. It's possible to avoid these problems by loading POPHELP after such TSRs. To avoid one common problem, POPHELP automatically detects when it is loaded before the Novell NetWare shell and refuses to pop up in this situation.

Using POPHELP

Once POPHELP is installed you can go about your business. The most likely place for you to start using it is within your text editor or within the TURBO integrated environment.

If you want to use a particular **Object Professional** routine and you can remember its name but not its parameters, position the editor cursor within or immediately after the procedure name you've typed in. Then press the Lookup hot key, <LeftShift><F1> by default. POPHELP searches the list of topics looking for an exact match (disregarding case). If that fails, it looks for a topic whose name starts with the search string.

If POPHELP can find a matching topic with one of these two search strategies, it will pop up a help window showing you the details about that routine. If it can't, it will instead pop up a window with the names of all the **Object Professional** units.

If you're not sure of the name of a routine, but you remember which unit it's in, press the Index hot key, <LeftShift><F3> by default, and you'll get a pick list showing all the **Object Professional** units. Select the unit you want and a list of topics within that unit will appear.

Within the help window you can use the arrow keys to move the cursor to highlighted areas which indicate links to other topics. Once the cursor is positioned within a highlighted region of interest, press <Enter> to bring up that topic. In this way you can browse around the help database at will. Note that the <Tab> and <ShiftTab> keys make the cursor jump immediately from link to link.

Once you've seen the information you want, you can return to the underlying application by pressing <Esc>.

If you want to see the same help topic again, press the Previous Topic hot key, <LeftShift><F2> by default.

POPHELP is based on the ScrollingHelpWindow object of OPHELP. By using this object (as opposed to PagedHelpWindow), POPHELP can provide line-by-line and horizontal scrolling of the help text, fully adjustable window size, and the ability to move the cursor freely around the help window.

Pasting Help Text into your Editor

Among its other capabilities, POPHELP also can paste marked blocks of help text into your editor. Cut/paste works much the same way that block copying works in the TURBO integrated environment.

The first step is to position your editor's cursor at the exact location where you want to paste the help text. Next, pop up the help screen and select your topic. Move the cursor through the text until it is positioned on the first character that you want to paste. Mark the beginning of the block by pressing <F7> (or <CtrlK><B>). Scroll through the text again until the cursor is positioned just beyond the last character that you want to paste. Press <F8> (or <CtrlK><K>) to mark end of the block. POPHELP will highlight the block. Finally, press <F4> (or <CtrlK><C>) to copy the marked block into your editor. POPHELP will immediately pop down and start feeding keystrokes to the underlying application. The next time you invoke POPHELP, the block will be hidden. If you want to toggle the block's visibility at any time, press <CtrlK><H>.

If the pasted text causes your editor to scroll, the display may temporarily look pretty strange since POPHELP can type much faster than the editor was probably designed to accept. However, your editor will "catch up" when POPHELP finishes pasting characters.

If your editor has an automatic indentation feature you will probably want to disable it–e.g., <CtrlQ><I> toggles autoindent in the TURBO IDE–during the paste. If autoindent is on during pasting, the pasted text may gradually move to the right of the screen.

If your editor interprets control characters entered directly (as does any WordStar-compatible editor), you'll want to avoid pasting any help text with such characters. In OPRO.HLP, only the line drawing examples contain control characters.

If the pasted block is larger than POPHELP's internal paste buffer (512 keystrokes), POPHELP must swap itself into memory to refill the buffer each time it is exhausted. When this occurs the pasting will be temporarily delayed until the buffer is refilled and POPHELP swaps out again. Depending on the speed of your system and whether POPHELP swaps to EMS or disk, this delay may range from a fraction of a second to a few seconds. If POPHELP cannot safely swap itself in when the buffer empties, the paste will be aborted prematurely. This should not occur when POPHELP is used over modern programmer's editors; EDLIN may have trouble.

POPHELP Command Summary

The following table summarizes the POPHELP commands we've discussed so far as well as a few commands we haven't mentioned yet.

<F1>
Displays the master topic index.

<AltF1>
Displays the topic that was displayed just prior to the current one. The cursor is positioned to its last location in the help window.

<F3>
Prompts for a new help file. A default extension of HLP is added to the name you enter. POPHELP will read any help file compiled using the **Object Professional** 1.0 MAKEHELP utility. Note that each help file needs memory space that is proportional to the help file's size and number of topics. POPHELP reserves a certain amount of memory when it goes resident and it cannot increase that amount later. Therefore, if you expect to load multiple help files during one POPHELP session, you should load the largest of them when POPHELP first goes resident.

POPHELP is configured with a maximum of 150 cross-references per topic. This limit is large enough to handle OPRO.HLP, but may not be adequate for larger help files. POPHELP simply ignores items in excess of the limits.

<F4>, <CtrlK><C>
If a block is marked and visible, pastes it from the help system to the underlying editor.

<F5>, <AltZ>
Toggles the zoom mode of the window.

<F6>, <AltM>
Enters move/resize mode–you can use the cursor keys to move the window around the screen, or the Shift-cursor keys to resize the window. Note that the window cannot be resized while it is zoomed.

<F7>, <CtrlK><B>
Marks the beginning of a block.

<F8>, <CtrlK><K>
Marks the end of a block.

<CtrlK><H>
Hides or unhides the currently marked block. <F4> will paste only if a block is visible.

<PgUp>, <PgDn>
Moves to the next or previous page of the current help topic.

<Left>, <Right>, <Up>, <Down>
Moves the cursor around the help screen. Attempting to move the cursor beyond the edge of the window will cause more text to scroll into the window.

<Tab>
Jumps to the next cross reference topic, if any.

<ShiftTab>
Jumps to the previous cross reference topic, if any.

<Enter>
Selects the cross-reference topic at the cursor location, if any. The new topic is loaded into the window and the cursor positioned on the topic's first character.

<Esc>
Exits POPHELP. The current topic and page are retained and may be redisplayed by pressing the Previous Topic hot key, <LeftShift><F2> by default.

Modifying the Object Professional Help File

We've supplied the source text for the **Object Professional** help file in case you want to make additional notes in it or to document routines of your own.

OPRO.HLP is built from a number of source files: OPRO.TXT is the main file and other files matching OPRO.T* are include files. The main file documents which units are to be found in each include file. It also documents many of the conventions used in designing the help file.

You can modify the source files using any text editor. See Section 4.G of this manual for more information.

F. DESKPOP: Desktop Manager

Of all the demonstration programs included with **Object Professional**, the one that best exemplifies its capabilities is DESKPOP. DESKPOP is a memory resident "desktop manager" that is comparable in many respects to Borland's SideKick Plus. DESKPOP provides you with a file manager, a text editor, a file browser, a calculator, a calendar, a quick reference chart, and a phone book/modem dialer--all in a single TSR that uses only 6K of RAM!

Although it is designed to run only in text mode, DESKPOP's user interface is similar to that of Microsoft Windows and the OS/2 Presentation Manager. All its component parts are displayed on the desktop as windows that can be moved, zoomed, and resized. Multiple windows may be opened at once, and you can switch from one to another at the press of the key. Or, if you prefer, you can use a mouse, because DESKPOP provides complete mouse support.

DESKPOP is also fully configurable. Using its companion program, DPINST, you can change its hotkeys, its colors, its key assignments--just about anything we could think of that you might want to change. And even if we forgot something, you can still change it, because the complete source code for the program is provided as well.

The next section explains how DESKPOP works and how you can use it. The following section covers DPINST.

1. DESKPOP

To load DESKPOP, enter

```
DESKPOP [Options]
```

at the DOS command line, where Options are one or more of the following:

```
/U        Unload DESKPOP from memory. If DESKPOP is not already installed, or it
          is not safe to unload, the program will abort with an error.
/E        Don't use EMS. If this option is not specified, DESKPOP will try to
          use EMS, if available, to store its swap buffers.
/M        Suppress swapping messages.
/K        Don't use (Kill) mouse. This option is especially useful if your mouse
          driver takes a long time to initialize the mouse hardware.
/Ppath    Specify the Path in which to store swap files. Path must name a
          directory, not a file.
/?        Display help screen.
```

Note that '-' may be used instead of '/' when specifying command line options.

Once DESKPOP has gone resident, it may be invoked with either of two hotkeys, <Alt><LeftShift><Tab> or <Ctrl><RightShift><D>. (The hotkey assignments may be changed using DPINST.)

Although DESKPOP takes up only 6K of RAM after it has gone resident, it actually allocates all available memory to itself beforehand to allow itself plenty of workspace once it is popped up. (See the documentation for OPSWAP for a detailed explanation of how swappable TSRs work.) At a minimum, DESKPOP requires approximately 415K of RAM to be available when it is loaded.

The Main Menu

When you activate DESKPOP using one of the two hotkeys, your screen will be divided into three parts. The top line of the screen shows DESKPOP's main menu, which we'll get to in a

moment. The bottom line is a status line. It is used to display reminders about key assignments, error messages, and prompts. Everything in between is "the desktop," the area where windows are displayed.

The main menu looks something like this:

```
Files Browser Calculator Editor caLendar Phonebook Reference Help
```

Each of these choices corresponds to a window that may be opened on the desktop. To open one of the windows, simply move the highlight to the appropriate choice and press <Enter>, or click on it with the left mouse button. When multiple windows are open, you can also use the main menu to activate an inactive window.

These are your choices:

Files
File manager. Used to display directories, copy/move/delete/rename files, etc.

Browser
File browser. Used to view ASCII or binary files of any size.

Calculator
A programmer's calculator with five modes of operation.

caLendar
A calendar that can display any month from January 1600 to December 3999.

Phonebook
A personal phone book and modem dialer.

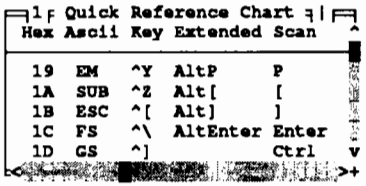
Reference
A programmer's quick reference chart.

Help
A simple help facility that shows the commands available for each window.

The main menu can be invoked from almost anywhere in the program by pressing <F10>.

Windows on the Desktop

With the exception of the Help window, each of the windows on the desktop looks something like this:



The number at the top left corner of the border is a window number. You can use this number to jump to a particular window quickly: e.g., <Alt2> would activate window #2. If you have a mouse, you can click the left mouse button on this "hot spot" to activate the window's local menu. (See the next section.)

The up-and-down arrow at the top right corner is a "zoom button." Clicking the left mouse button on this hot spot will zoom or unzoom a window. (The Calculator and Calendar windows do not have a zoom button.)

Between these two hot spots is the window's title, which is treated as one big hot spot (or "hot region"). By clicking the left mouse button anywhere on the title, you activate a mode in which the mouse can be used to drag the window across the desktop. Click the left mouse button a second time when you are finished.

The plus sign ('+') at the bottom right corner of the border is also a hot spot. Clicking the left mouse button on it activates a mode in which the mouse can be used to adjust the size of the window. Click the left mouse button a second time when you are finished. (Note that the Calculator and Calendar windows cannot be resized.)

If the window is scrollable, it will have scroll bars on the right and bottom edges of its border, as shown here. These can be used to jump from one place to another rapidly.

The mouse may also be used to activate an inactive window. If you can see any part of an window (other than its shadow), you can activate it immediately by clicking the left mouse button on it. Similarly, clicking the left mouse button anywhere on the top row of the screen will activate DESKPOP's main menu. If the mouse cursor is on top of one of the menu choices at the time, that choice will be highlighted, but not selected. (You can select it by clicking the left mouse button a second time.) If it is not on one of the menu choices, the previously selected item will be highlighted, just as it would if you pressed <F10>.

When a new window is opened, it is placed on the desktop just on top of the previously active window. Once a window has been opened, it will retain the same window number until it is closed. If a window in the middle of the sequence is closed, the next window opened will be assigned its number (or the lowest available number, if more than one have been closed).

Normally, when you switch to the "next" window (using <F6>), the window with the next highest number will be activated or, if there isn't one, the first window. Note, however, that the window manager doesn't really pay attention to window numbers. If you activate an open but inactive window in the middle of the stack--using <Alt-n>, where n is its window number, or by clicking on it with the left mouse button, or by selecting it from the main menu--that window will be pulled to the top of the stack. For example, if the windows were previously stacked in the order 1-2-3-4, with window 4 at the top, and you activated window 2 by pressing <Alt2>, the new order would be 1-3-4-2.

Local Menus

As indicated earlier, each of the windows has a "local menu" that can be activated either by pressing <F9> or by clicking the left mouse button on the window number at the top left corner of its border. The contents of these menus vary somewhat from one window another, but they all display at least four choices:

Close does just what its name suggests: it lets you close the active window.

Move activates a mode in which the cursor keys may be used to drag the window across the desktop to a new location. When you are finished, press <Enter>.

Similarly, Resize lets you adjust the size of the active window using the cursor keys. Press <Enter> when you are finished.

Zoom lets you temporarily adjust the size of a window so that it fills the desktop. If the window is already zoomed, it is instead restored to its previous size ("unzoomed").

Note that the Resize and Zoom choices are disabled for two of the windows, Calculator and Calendar, because their sizes are fixed.

Additional choices available in a given window's local menu are described below in the appropriate section.

Hotkeys

All the functions just described may be performed using the keyboard as well as the mouse. Within all of the windows that can be placed on the desktop (except the Help window), the following hotkeys can be used:

<AltM>
Move active window. Use the cursor keys to drag the window to the desired position, then press <Enter>.

<AltR>
Resize active window. Use the cursor keys to adjust the size of the window to the desired position, then press <Enter>.

<F5>, <AltZ>
Zoom/unzoom active window.

<F6>
Activate next window.

<CtrlF6>
Activate previous window.

<Alt1>..<Alt8>
Activate window 1..8.

<F9>
Invoke local menu.

<F10>
Invoke main menu.

<AltX>
Exit from DESKPOP.

<AltU> <AltU>
Unload DESKPOP.

<AltS>
Show available memory.

Files

DESKPOP's file manager is based on the DirList object implemented in OPDIR. See the documentation for that unit for details on how a multiple-choice directory list (one that allows block marking) works.

When the Files window is active, the status line will look something like this:

```
<F3> New mask  <F9> Menu | Disk: 2,258,944    Files: 58      Size: 3,042,097
```

The "Disk:" value shows you the amount of space remaining on the drive containing the files displayed in the window. The "Files:" value shows you the number of files being displayed (directories are not included). And the "Size:" value shows you their collective size.

In addition to the standard directory list commands, the Files window also provides the following commands:

<F3>
Specify a new file mask.

<Space>
Toggle between Name mode and All mode. In Name mode, only the names of matching files and directories are displayed. In All mode, their sizes, date-time stamps, and file attributes are displayed as well.

<CtrlK> <C>
Copy marked files to a specified drive:\directory. If no files are marked, the currently highlighted file will be copied.

<CtrlK> <V>
Move marked files to a specified drive:\directory. If no files are marked, the currently highlighted file will be moved. Files can be moved to another drive as well as to another directory.

<CtrlK> <Y>
Delete marked files/directories. If no files are marked, the currently highlighted file will be deleted. Note that directories may be deleted only if they are empty.

<CtrlY>
Delete highlighted file/directory. Note that directories may be deleted only if they are empty.

<CtrlK> <N>
Rename highlighted file.

<AltE>
Load highlighted file into Editor. The Editor window will be opened automatically if it is not open already. If it is open, and the file being edited has been modified, you will be asked if you want to save it. If you answer Yes, the file will be saved before the new one is loaded. If you answer No, the edits to the current file will be lost. If you press <Esc>, the command will be cancelled.

<AltB>
Load highlighted file into Browser. The Browser window will be opened automatically if it is not open already.

Besides the usual Close/Move/Resize/Zoom window options, the local menu for the Files window offers the following choices, several of which correspond to standard keyboard commands. Where applicable, the corresponding command is shown to the right of the menu choice:

Directory

Change
Change directories. This command does *not* change the directory displayed in the window. It simply changes the current DOS directory.

Make
Make a new directory.

Remove
Remove a specified directory. If the currently highlighted item is a directory, its name will be the default response.

Files

Attributes
This submenu lets you change the file attributes for all marked files. If no files have been marked, the operation will be performed on the currently highlighted file. The first four choices–Archive, Hidden, Read-only, and System–specify the desired attribute settings. When you have set these options the way you want, select "Go" to perform the operation.

Browse **<AltB>**
Load highlighted file into Browser.

Copy **<CtrlK> <C>**
Copy marked files.

Delete **<CtrlK> <Y>**
Delete marked files/directories.

Edit **<AltE>**
Load highlighted file into Editor.

Move **<CtrlK> <V>**
Move marked files.

Rename **<CtrlK> <N>**
Rename highlighted file.

Options

Confirm
If this option is on ('Yes'), as it is by default, you will be asked to confirm all file-related operations (copy, delete, move, etc.). When prompted for confirmation, you may answer Yes (perform the operation), No (don't perform the operation), Quit (abort the operation), or All (perform the operation on all marked files without asking).

Format **<Space>**
Toggle between Name mode and All mode.

Sort
Change the order in which file/directory names are sorted.

Browser

DESKPOP's Browser window is based on the Browser object implemented in OPBROWSE. See the documentation for that unit for details on how a file browser works.

When the Browser window is active, the status line will look something like this:

```
<F3> New file  <F9> Menu  <AltH> Hex/Ascii  <Esc> Close  | Line 1331  Col 1
```

The numbers on the right indicate the number of the line at the top of the window, and the column for the character displayed at the left edge of the window. When the Browser is reading from the disk or searching, a "Working..." message will be displayed in their place.

Besides the usual Close/Move/Resize/Zoom window options, the Browser's local menu offers the following choices, several of which correspond to standard keyboard commands. Where applicable, the corresponding command is shown to the right of the menu choice:

Block

Hidden **<CtrlK> <H>**
Toggle the display of marked blocks.

Print **<CtrlK> <P>**
Print the currently marked block.

Write **<CtrlK> <W>**
Write the currently marked block to a file.

Options

Expand tabs **<CtrlQ> <T>**
Toggle tab expansion (ASCII mode only). If this option is on, tabs are expanded to spaces on 8-column boundaries.

High bits **<CtrlQ> <H>**
Toggle stripping of high bits.

Mode **<CtrlH>, <AltH>**
Toggle between ASCII and hex modes.

Search

Again **<CtrlL>**
Repeat last find operation.

Find **<CtrlQ> <F>**
Search for any string of up to 30 characters.

Options
These choices let you change the default search options.

> **Global**
> Search globally?
>
> **Ignore case**
> Ignore case when searching?
>
> **Reverse**
> Search backwards?

Text markers

Jump **<CtrlQ> <0>..<CtrlQ> <3>**
Jump to text marker 0..3. You will be prompted for the marker to jump to.

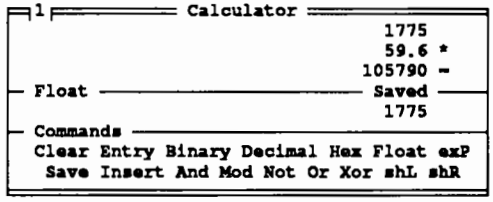
Set **<CtrlK> <0>..<CtrlK> <3>**
Set text marker 0..3. You will be prompted for the marker to set.

Calculator

The Calculator in DESKPOP is designed especially for programmers. It can operate in any of 5 different modes--2 for floating point calculations, 3 for integer calculations--and can perform 11

different arithmetic and logical operations. Integer calculations are performed using long integers in the range -2,147,483,648 ($80000000) to 2,147,483,647 ($7FFFFFFF). Floating point calculations are performed using BCD (Binary Coded Decimal) reals with 18 digits of precision and a range of 1E-63 to 1E+63.

The Calculator window looks something like this:



The top three lines, the calculation window, are used for entering numbers and displaying the results of calculations. The second of the two is always used for entering numbers, and the third is used to display results. To perform a calculation, follow these steps:

1. Enter the first number.
2. Press the appropriate key for the arithmetic or logical operation (+, -, *, /, A, M, N, O, X, L, or R).
3. Enter the second number.
4. Press <Enter> or < = > to display the result.
   - or -
   Press one of the keys listed in step 2 to complete the calculation and use the result as the first value in a second calculation.

Once you have a result, you can either start a new calculation by entering a number or press one of the keys listed in step 2 to use the result as the first value in a second calculation. Note that it is not necessary to press <NumLock> before entering numbers on the numeric key pad. The Calculator window automatically turns NumLock on when it is activated, and restores the previous state of NumLock on exit.

The bottom two lines of the inner window show a menu of available commands. Just above it is the value in the active Save Buffer. There are two such buffers, one for floating point values and one for integers.

The following commands are available in the Calculator:

Basic Calculator Commands

<C>
Clear all. Clear the entire calculation window.

<E>
Clear Entry. Clear the current Entry (second line of calculation window).

<S>
Save. Save the current entry in the active Save Buffer.

<I>
Insert. Make the number in the active Save Buffer the current entry.

Mode Commands

<B>
Binary. Switch to binary mode (integer).

<D>
Decimal. Switch to decimal mode (integer).

<H>
Hexadecimal. Switch to hexadecimal mode (integer).

<F>
Floating point. Switch to regular floating point mode (real). Results are displayed without exponents whenever possible--e.g., '0.0057' rather than '5.7E-03'.

<P>
exPonential. Switch to exponential floating point mode (real). Results are always displayed in exponential format--e.g., '5.7E-03' rather than '0.0057'.

Arithmetic Operators

<+>
Add Value1 and Value2.

<->
Subtract Value2 from Value1.

<*>
Multiply Value1 by Value2.

</>
Divide Value1 by Value2.

<M>
Mod. Remainder after dividing Value1 by Value2 (integer modes only).

Logical operators (integer modes only):

<A>
And. Do a bit-wise AND of Value1 with Value2; result has all bits that are set in both Value1 and Value2.

<N>
Not. Do a bit-wise negation of the current entry.

<O>
Or. Do a bit-wise OR of Value1 with Value2; result has all bits that are set in Value1 or Value2.

<X>
Xor. Do a bit-wise eXclusive OR of Value1 with Value2; result has all bits that are set in Value1 or Value2 but not both.

<L>
shL. Shift Value1 left by Value2 bits. If Value2 is greater than 31 or less than 0, the result is 0.

<R>
shR. Shift Value1 right by Value2 bits. If Value2 is greater than 31 or less than 0, the result is 0.

Other Commands

<BkSp>
Delete previous character.

<CtrlA>..<CtrlF>
When in hexadecimal mode, these keys can be used to enter the values $A through $F, respectively.

<Esc>, <CtrlBreak>, <ClickRight>
Close window.

<F1>, <ClickBoth>
Request help.

<AltI>
Insert a number into the file in the Editor window (which must be open). If a calculation has just been finished, the result of the calculation will be inserted, otherwise the number currently being entered (Value2). If the current calculation mode is floating point or exponential, you can have the number inserted in the form of a BCD typed constant. For example, the string generated for 1.2345 would be '($40,$00,$00,$00,$00,$00,$00,$50,$34,$12)'.

Editor

DESKPOP's Editor is based on the TextEditor object implemented in OPEDITOR. See the documentation for that unit for details on how a TextEditor works.

When the Editor window is active, the status line will look something like this:

```
<F2> Save file  <F3> New file  <F9> Menu  <Esc> Close  ° Line 1340  Col 1
```

The numbers on the right indicate the current position of the cursor. When a search operation is in progress, the line counter will be updated frequently to tell you how far the search has progressed.

In addition to the standard editing commands, the Editor also allows you to use <Esc> or <ClickRight> to close the edit window. If the file being edited has been modified, you will be asked if you want to save it before closing the window. Pressing <Esc> or <ClickRight> at this prompt will cancel the command.

Besides the usual Close/Move/Resize/Zoom window options, the Editor's local menu offers the following choices, several of which correspond to standard keyboard commands. Where applicable, the corresponding command is shown to the right of the menu choice:

Block

Change case
The choices change the case of an entire block. If no block is marked, or if the block is hidden, only the case of the character at the cursor will be changed.

Lower **<CtrlO><V>**
Convert block to lower case.

Toggle **<CtrlO><O>**
Toggle the case of all characters in block

Upper **<CtrlO><U>**
Convert block to upper case.

Hidden **<CtrlK><H>**
Toggle the display of marked blocks.

Indentation

Indent **<CtrlK><I>**
Indent marked block by N (indentation level) columns.

Unindent **<CtrlK><U>**
Unindent marked block by N (indentation level) columns.

Level **<CtrlO><B>**
Set indentation level (defaults to 2).

Print **<CtrlK><P>**
Print the currently marked block.

Read **<CtrlK><R>**
Read a file into the Editor at the position of the cursor.

Write **<CtrlK><W>**
Write the currently marked block to a file.

File

New **<F3>**
Load a new file into the Editor.

Save **<F2>, <CtrlK><S>**
Save the file being edited.

Write as **<CtrlK><N>**
Save the file being edited under a new name.

Options

Autoindent **<CtrlO><I>**
Toggle auto-indent mode.

Backups
If this option is on, the Editor will create a backup (BAK) file when saving a file that already exists.

Delete joins
If this option is on, pressing <Del> at the end of a line will join the current line with the following line.

Margin **<CtrlO><R>**
Set the right margin.

Partial files
Load partial files? If this option is on, and a new file is too large to be read in its entirety, the Editor will read in as much of it as possible. This setting does not affect the block read command.

Smart tabs **<CtrlO><F>**
Toggle smart tabs on or off.

Tab size **<CtrlO> <T>**
Set tab size (fixed tabs only).

Word wrap **<CtrlO> <W>**
Turn word wrap on or off.

Search

Again **<CtrlL>**
Repeat last find/replace operation.

Find **<CtrlQ> <F>**
Search for any string of up to 30 characters.

Options
These choices let you change the default find/replace options.

> **Block only**
> Search only within currently marked block?
>
> **Confirm**
> Confirm replacements?
>
> **Global**
> Search globally?
>
> **Ignore case**
> Ignore case when searching?
>
> **Reverse**
> Search backwards?

Replace **<CtrlQ> <A>**
Replace a string of up to 30 characters with another string.

Text markers

Hidden **<CtrlK> <M>**
Toggle display of text markers

Jump **<CtrlQ> <0>..<CtrlQ> <3>**
Jump to text marker 0..3. You will be prompted for the marker to jump to.

Set **<CtrlK> <0>..<CtrlK> <3>**
Set text marker 0..3. You will be prompted for the marker to set.

Calendar

DESKPOP's Calendar is based on the PickList object implemented in OPPICK. See the documentation for that unit for details on how a PickList works.

In addition to the standard pick list commands, the Calendar also provides the following commands:

<Home>
Cursor to beginning of month.

<End>
Cursor to end of month.

<PgDn>, <CtrlC>
Display next month.

<PgUp>, <CtrlR>
Display previous month.

<CtrlPgDn>, <CtrlQ><C>
Display next year.

<CtrlPgUp>, <CtrlQ><R>
Display previous year.

<CtrlHome>
Display current month; move cursor to today's date.

<Esc>, <ClickRight>
Close window.

Note that the Calendar window is of a fixed size. It can be moved, but it cannot be zoomed or resized.

Phone Book

Like the Calendar, DESKPOP's Phone window is based on the PickList object implemented in OPPICK. See the documentation for that unit for details on how a PickList works.

When you first open the Phone window, you will be asked for the name of a phone book, a file with a default extension of "PB". If the file does not exist, you will be asked if you want to create it. If you wish, you can ask to open the supplied sample phone book, PHONE.PB. Assuming that you did so, the inner portion of the Phone window would look something like this:

```
Index   Name                          Phone number
-------------------------------------------------
Bor   | Borland                     | 1-408-438-5300
TPS 1 | TurboPower Software         | 1-408-438-8608
TPS 2 | TurboPower Software         | 1-800-333-4160
```

The row at the top consists of column headers, and the second row is simply a divider. The remaining rows show selected portions of each entry in the phone book.

The Index column shows the abbreviation associated with a given entry. To get to that entry quickly, you can enter the characters that make up the abbreviation in sequence: pressing <B> would take you to the first entry with an index that starts with "B", pressing <O> would then take you to the first entry starting with "BO", and so on. The entries are always sorted on the basis of the Index field.

The actual contents of the second column depend on what was entered in the phone book. If both a first and last name were entered, they will be displayed here in "LastName, FirstName" format. If only one of the two was entered, it will be displayed here by itself. If neither was specified, but a company name was, the company name will be displayed here. If all three fields in the entry are empty, this column will be blank. The complete contents of a phone book entry are described below.

The third column shows the corresponding phone number. If you are planning on using your modem to dial phone numbers, the phone number should include all the digits required to dial the number (long distance prefix, area/country code). Note that the individual digits may be

separated by dashes ('1-234-555-1212'), spaces ('1 234 555 1212'), and/or parentheses ('(123) 555-1212'); these characters will be ignored by the modem.

In addition to the standard pick list commands, the Phone window also provides the following special commands:

<Enter>
Edit the highlighted entry (see below for editing instructions). If there are no entries in the phone book, this command can be used to add a new one.

<ClickLeft>
Select an entry for editing using the mouse.

<CtrlN>
Insert a new entry into the phone book.

<CtrlY>
Delete the highlighted entry.

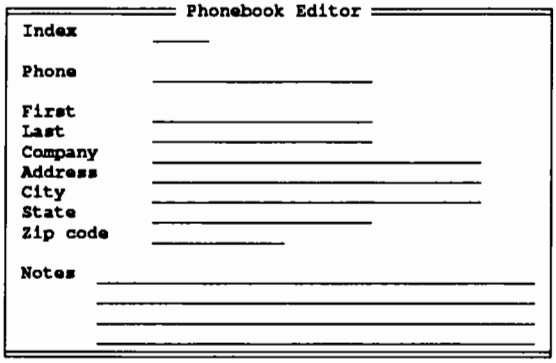
<F3>
Load a new phone book.

<AltD>
Dial the phone number in the highlighted entry using the modem. When you hear your modem's speaker indicating that the phone at the other end is ringing, pick up the receiver and press any key to hang up the modem. If the phone is busy, press a key without picking up the receiver. Be sure to wait until the modem is finished dialing before pressing a key. Your modem *must* be Hayes-compatible for this command to work. If you have difficulty using this command, you may need to use DPINST to change one of the settings that affect the way DESKPOP communicates with the modem (baud rate, COM port, dial string, hang up string). Doing so will be absolutely necessary if you have a 300 baud modem or if you don't have a touch tone phone.

<AltH>
Hang up the modem. This command is needed only if you gave the dial command and accidentally pressed a key before the modem was finished dialing.

<Esc>, <ClickRight>
Close the window.

When you select an entry for editing or ask to add a new entry, a second window will be displayed that looks something like this:



This window is a data entry screen created with OPENTRY. (See the documentation for that unit for details on how an EntryScreen works.)

The fields in the entry screen are fairly self-explanatory, except for the first and last. The purpose of the Index field is described above. You may enter whatever you want here, but the field may not be left empty. (Phone is also a "required field.") The last field, Notes, may be used to enter miscellaneous information: birthdays, anniversaries, alternate phone numbers, etc.

Quick Reference Chart

DESKPOP's Quick Reference Chart takes the idea of SideKick's ASCII chart a few steps further. It displays a series of translation tables for values in the range 0-255. These tables show

- the value in decimal, hexadecimal, and binary notation
- the corresponding ASCII character
- the corresponding ASCII code--e.g., 7 = "BEL"
- the corresponding BIOS key codes--e.g., 16 = <CtrlP> or <AltQ> (extended scan code)--including those keys that are available only if the OPENHKBD unit is being used (displayed in high intensity or white), as well as the pseudo-scan codes returned by certain routines in the OPMOUSE unit (displayed in high intensity or yellow)
- the corresponding hardware scan code
- the corresponding video attribute for both monochrome and color display adapters

The following commands are available in the Quick Reference Chart:

<Up>, <CtrlE>, <CtrlW>
Scroll window up.

<Down>, <CtrlX>, <CtrlZ>
Scroll window down.

<Left>, <CtrlS>
Scroll window left 1 column.

<Right>, <CtrlD>
Scroll window right 1 column.

<CtrlLeft>, <CtrlA>
Scroll window left 10 columns.

<CtrlRight>, <CtrlF>
Scroll window right 10 columns.

<PgUp>, <CtrlR>
Scroll window up one page.

<PgDn>, <CtrlC>
Scroll window down one page.

<Home>, <CtrlQ><S>
Scroll to left edge of chart.

<End>, <CtrlQ><D>
Scroll to right edge of chart.

<CtrlPgDn>, <CtrlQ><C>
Scroll to end of chart.

<CtrlPgUp>, <CtrlQ><R>
Scroll to top of chart.

<Esc>, <CtrlBreak>, <ClickRight>
Close window.

<F1>, <ClickBoth>
Request help.

<CtrlB>
Toggle attribute blinking. Normally, any video attribute with a value greater than 127 (01111111 in binary) will produce blinking text. Since blinking text can be quite annoying if not used in moderation, the Quick Reference Chart suppresses blinking in all of its color samples when attribute blinking is enabled; to indicate attributes that produce blinking, it instead highlights the names of the foreground and background colors. If attribute blinking is disabled, it displays color samples normally. Note that the <CtrlB> command reprograms the CRT controller, and DESKPOP does not restore it to its previous state when you leave the program.

Help

The Help window may be activated at any time by pressing either <F1> or <ClickBoth>. The help facility is not, as you might expect, a context-sensitive help system created with OPHELP. Rather, it is a file browser (much like the Browse window) that automatically finds and loads an ASCII text file called DESKPOP.HLP. You can then scroll through the help file in the same manner that you would examine a file in the Browse window.

DESKPOP looks for its help file first in the current directory, then in the directory from which it was loaded (under DOS 3.x or higher), then in the directories on DOS's PATH. If it cannot find it in any of these places, you'll see a message indicating that the help file was not found.

DESKPOP.HLP simply contains a list of all the commands available in each of the windows and the keys that they are assigned to. When you change these key assignments using DPINST, DPINST will create a new copy of DESKPOP.HLP that reflects your changes.

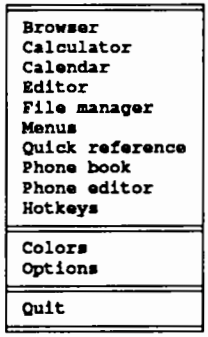
Unlike most of the other windows that may be placed on the desktop, the Help window cannot be zoomed, moved, or resized. Nor can you switch to another window while the Help window remains open. When you are through examining the help file, press <Esc> or <ClickRight> to close the Help window and return to what you were doing. If you want to be able to keep the help file open while working with other windows, you can load it into the Browser or Editor.

2. DPINST

DPINST allows you to customize DESKPOP to suit your personal preferences concerning things like hotkeys, colors, and key assignments. Before running it, be sure that both DESKPOP.EXE and DESKPOP.HLP are in the current directory. Then enter

```
DPINST
```

at the DOS prompt. DPINST will require a few moments to locate all the configuration data in DESKPOP.EXE. When it is finished, you'll see the following menu:



The menu choices in the top section allow you to customize the key assignments for the different parts of the program. The Colors choice allows you to change the video attributes used by DESKPOP. The Options choice allows you to change various settings that affect DESKPOP's behavior.

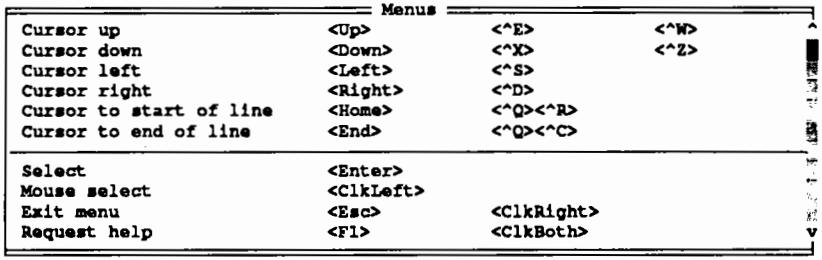
Quit returns you to the DOS prompt. If you have made any changes, DPINST will first ask if you want to save them to disk. Press <Y> to save the changes and exit, <N> to ignore the changes and exit, or <Esc> to cancel the quit command. If you answer Yes, DPINST will update the configuration data in DESKPOP and write a new DESKPOP.HLP file that reflects any changes you made to the key assignments.

Note that if DESKPOP is loaded at the time you run DPINST, you must unload it and then reload it in order for your changes to take effect.

Changing Key Assignments

As we said, all the choices in the first section of the menu pertain to key assignments. Since the same procedure is used to change them in each case, we'll concentrate on the procedure in general rather than looking at the specific cases individually.

If you selected the Menus choice, for example, you'd see something like this:



The first column shows the name of each command, and the next three columns show the various key sequences assigned to that command.

To change a key assignment or add a new one, use the cursor keys to move the highlight bar to the one you wish to change and press <Enter>, then press the key(s) you want to assign the command to.

While entering a key assignment, the following commands are available:

<BkSp>, <CtrlBkSp>
Delete last key.

<CtrlC>, <CtrlY>
Clear the key assignment.

<CtrlR>, <CtrlQ><L>
Restore the previous value.

<Enter>, <ClickLeft>
Accept key assignment.

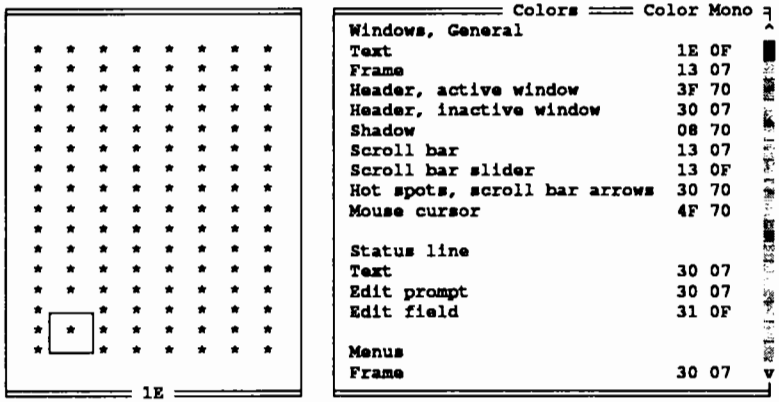
<Esc>, <CtrlBreak>, <ClickRight>
Cancel the operation.

If you need to enter one of these keys as part of the key assignment, press <ScrollLock> or <ClickBoth> to switch to literal mode rather than command mode, enter the key(s), then press <ScrollLock> to switch back to command mode.

To accept any changes that you have made and return to the main menu, press <Esc>. DPINST will then check to make sure that there are no conflicts among the key assignments. If any conflicts are found, an error message will be displayed and the keys causing the conflicts will be highlighted. DPINST will not allow you to return to the main menu until you have resolved the conflict(s). If you decide that you'd rather just leave the keys the way they were before you made your changes, you can press <CtrlR> to Restore the default settings, then press <Esc>.

Changing Colors

If you were to select the Colors choice from the main menu and then immediately press <Enter>, you'd see something like this:



The window on the right lists the various video attribute settings. Each attribute has two settings, one for color video modes and one for monochrome modes. You change a particular setting by moving the highlight to it and pressing <Enter>, or by clicking the left mouse button on it. The window on the left, which shows all the colors you have to choose from, will then draw a box around the current setting, as shown above. Using the cursor keys, move the box to your preferred setting and press <Enter>. You can also perform this operation with the mouse. Just drag the mouse across your desk until the box is in the right place, then click the left mouse button.

When you are finished changing colors, press <Esc> to return to the main menu.

Changing Options

When you select the Options choice from the main menu, a scrollable data entry screen will be popped up that lets you change a variety of settings. (If you need instructions on how to use an **Object Professional** entry screen, see the documentation for OPENTRY.) When you are finished making changes, press <Esc>, <ClickRight>, or <CtrlEnter> to return to the main menu. Note that pressing <Esc> or <ClickRight> will *not* cancel all the changes you made, only your changes to the field being editing at the time.

The fields in the entry screen are grouped into sections, and so are the descriptions of them that appear below. In each case the first line gives the name of the option setting, followed by its default value. The second and following lines give a brief description of the option.

TSR Related Options

Primary hotkey **$0A0F**
Primary key combination used to activate DESKPOP. Enter the appropriate value in hex, or press <F10> to select a key combination from a pair of pick lists (one for the shift flags, one for the scan code). See the documentation for OPTSR for a discussion of hotkey values.

Hotkey string **Alt+LeftShift+Tab**
String representation of primary hotkey. Displayed by DESKPOP just before going resident. If you have enough room, you can include the secondary hotkey assignment here if you wish.

Secondary hotkey **$0520**
Secondary key combination used to activate DESKPOP.

Path for swap files **C:**
Drive:\Directory in which to write swap files.

Swap name #1 **DESKPOP1.$$$**
Swap name #2 **DESKPOP2.$$$**
Names for the two swap files (one for DESKPOP's code and data, one for the code and data that will be overwritten when DESKPOP is swapped in).

Use EMS if available **Y**
Use EMS to store swap data if it is available?

Show swap message **Y**
Display "Swapping in..." and "Swapping out..." messages when popping up/down?

Display Options

Enable mouse support **Y**
Use the mouse in DESKPOP if one is installed?

Color selection **Auto**
Auto means choose color or monochrome attributes Automatically. Color means always use the set of attributes for a color system. Mono means always use the set for a monochrome system.

Browser Options

Default extension **<none>**
Default extension to use when prompting for filenames. (No default setting.)

Default mode **ASCII**
Default display mode (ASCII or Hex).

Tab expansion **Y**
Expand tabs to spaces?

Strip high bits **N**
Strip high bit of each character when displaying it?

File buffers **4**
Number of 4K buffers used to hold text from the file being displayed.

Editor Options

Default extension **<none>**
Default extension to use when prompting for filenames. (No default setting.)

Auto indent **Y**
Auto-indent mode on by default?

Delete joins lines **N**
Should pressing <Del> at end of line cause the following line to be joined with it?

Indent starts para **N**
When reformatting, should a space at beginning of line be considered start of a new paragraph?

Smart tabs **Y**
Smart tabs on by default (rather than fixed tabs)?

Tab size **8**
Number of columns between tab stops (fixed tabs only).

Word wrap **Y**
Word wrap on by default?

Wrap at column 1 **N**
When cursor is at column 1, should pressing <Left> move cursor to end of previous line?

Make backup files **Y**
When saving a file that already exists, should current file first be renamed to filename.BAK?

Read partial files **N**
If a file is too large to be read in its entirety, should the part of it that will fit be read in?

Buffer size **65521**
Size of the largest file that can be loaded into the Editor. (The actual limit is 2 less than this.)

File Manager Options

Buffer size **16384**
Indirectly determines how many file/directory names can be displayed within the Files window at once. 16K strikes a balance between capacity and memory usage. Lower it only if you are very short on memory. Raising it will limit the amount of memory available for other windows.

Confirm changes **Y**
Prompt for confirmation of all file operations?

Filename format **Name**
Format for filenames (Name or All).

Sort order **Name**
Default order in which to sort filenames (Name, Date, Size, or MS-DOS).

Phone Book Options

Default phone book **<none>**
Filename to display as default choice when the Phone window is opened. (No default setting.)

Default extension **PB**
Default file extension for phone books.

Modem installed **Y**
If 'N' (No), the dial and hang-up-modem commands in the phone book are disabled. Use N if you don't have a modem or it is not Hayes-compatible.

Communications port **COM1**
COM port that modem is attached to (COM1 or COM2).

Baud rate **1200**
Baud rate to set the modem to when dialing (300, 1200, or 2400).

Dial modem string **ATDT**
String sent to a Hayes-compatible modem to dial a phone number (ATDT = ATtention Dial Touchtone). If your phone line is not set up for Touch-Tone dialing, try "ATDP" instead.

Hang up modem string **ATH0**
String used to hang up modem.

G. Purchase Agreement

This software and accompanying documentation are protected by United States Copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

Object Professional is copyright (c) 1987-1991 by TurboPower Software. Portions are copyright (c) 1985, 1986 by Sunny Hill Software, and are used under license to TurboPower Software.

TurboPower Software authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no conditions may you remove the copyright notices made part of the software or documentation.

You may distribute, without runtime fees or further licenses, your own compiled programs based on any of the source code and precompiled units of **Object Professional**. You may not distribute any of the **Object Professional** source code, precompiled units (TPU files), or example programs without written permission from TurboPower Software.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend on **Object Professional** units. However, others who receive your units would need their own copies of **Object Professional** in order to write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. We expect that group programming projects making use of this software will purchase a copy of the software and documentation for each member of the group. Contact TurboPower Software for volume discounts and site licensing agreements.

With respect to the physical diskettes and documentation provided with **Object Professional**, TurboPower Software warrants the same to be free of defects in materials and workmanship for a period of 30 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software will replace the defective diskette(s) or documentation at no cost to you.

TurboPower Software warrants that the software will function as described in this documentation for a period of 30 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software will update the defective software at no cost to you.

TurboPower Software further warrants that the purchaser will remain fully satistfied with the product for a period of 30 days from receipt. If you are dissatisfied for any reason, and TurboPower Software cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software, we will refund the full purchase price of the software upon receipt of the original program diskette(s) and documentation in undamaged condition. TurboPower Software honors returns from authorized dealers, but cannot offer refunds directly to anyone who did not purchase the product directly from us.

TURBOPOWER SOFTWARE DOES NOT ASSUME ANY LIABILITY FOR THE USE OF **OBJECT PROFESSIONAL** BEYOND THE ORIGINAL PURCHASE PRICE OF THE SOFTWARE. IN NO EVENT WILL TURBOPOWER SOFTWARE BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS

OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

By using this software, you agree to the terms of this section. If you do not agree, you should immediately return the entire **Object Professional** package for a refund.

TurboPower Software offers telephone support for **Object Professional** on an as-available basis at no extra charge. You are welcome to call our support line on Monday-Friday 9am to 5pm (Mountain time) to ask for help. We also maintain a CompuServe account and public forum to support you. CompuServe access information and the technical support telephone number are listed on the title page of this manual.

We believe in supporting our customers and will do our best to make this software handle the job it was designed to do!

2. Principles of Organization

This chapter aims to give you a broad overview of the library as a whole. It is not a list of features, but rather an account of how the library is arranged and of the governing principles that shaped it. Subsequent chapters will describe individual units in detail; this one will try to give you an idea of how they fit together. The goal here is to help you see The Big Picture.

The chapter is divided into two parts. The first part arranges the units into logical groups based on the kinds of services they provide. The second part discusses the numerous conventions and principles that determined how they were written.

A. Overview of the Library

If you've looked at the Table of Contents, and you surely have, you know that the manual divides the library into three groups: "Screen Management," "Data Input/Output," and "TSRs and More" (i.e., everything that doesn't fit into the first two categories). It's organized this way to try to minimize the amount of searching you have to do to find a routine to perform a particular task. If you want to find out how to draw a bordered popup window on the screen, you shouldn't have to flip through three manuals to find all the information you need.

But from another perspective these groups are somewhat misleading. OPCRT and OPWINDOW both have screen-related routines, for example, but they are *very* different: OPCRT contains a lengthy series of low-level routines, while OPWINDOW contains a handful of objects that operate at a much higher level. So this overview proposes a different grouping, one that reflects the qualitative differences among the various units.

The first group might be referred to as the *system-oriented units,* units that are comprised largely if not entirely of traditional procedures and functions rather than objects, units that in many cases provide low-level routines that are in turn used by higher level objects in one of the other groups. The units in this group are:

```
OP8087
OPASCIIZ
OPBCD
OPCOLOR
OPCRT          (ColorSet)
OPDATE
OPDOS
OPEMS
OPENHKBD
OPERRHAN
OPEXEC
OPEXTMEM
OPINLINE
OPINT
OPMACED
OPMACRO
OPMOUSE
OPREPLAY
OPSORT
OPSTRDEV
OPSTRING
OPTSR
OPSWAP
OPSWAP1
```

The only unit that is really out of place in this list is OPMACED, which implements a keyboard macro editor. In terms of functionality, it would seem to belong in the third group; it appears in the first because it does not contain any objects. OPCRT is out of place here only in the sense that it implements the ColorSet object. (Names in parentheses indicate objects implemented in a given unit.)

The second group consists of *units that implement low-level objects* (or help to implement them). The units in this group are:

```
OPABSFLD    (AbstractField)
OPCLONE     (Cloner)
OPCMD       (CommandProcessor, CommandPacker)
OPDEVICE
OPFEDIT
OPFIELD     (SelectField, EntryField, various derived types)
OPFRAME     (ScreenRect, HeaderNode, ShadowNode, HotNode, ScrollBar, Frame)
OPLARRAY    (AbstractArray, RamArray, EmsArray, VirtualArray, OpArray)
OPPRINT     (PrintMode, Printer)
OPPRNLOW    (BasePrinter, DosPrinter, FlexiblePrinter, BiosPrinter)
OPROOT      (Root, SingleList, DoubleList, BufIdStream, Library, many others)
OPSEDIT     (SimpleLineEditor)
```

Most of the objects in these units form the building blocks for higher level objects: for example, the container classes in OPROOT (stacks, lists, sets); the field types in OPABSFLD and OPFIELD; the CommandProcessor in OPCMD; and the Frame, ScrollBar, and HotNode objects in OPFRAME. In other cases, these units provide objects that are useful in themselves, but which might be used as building blocks: for example, the Cloner object in OPCLONE, the large array objects in OPLARRAY, and the SimpleLineEditor in OPSEDIT.

If you look at the object hierarchy in Appendix C, you'll notice that most of the objects in these units are descended directly from the Root object defined in OPROOT.

The third group consists of *units that implement high-level objects* by building upon the resources provided by the units in the first two groups. The units in this group are:

```
OPBROWSE    (Browser)
OPCAL*      (Calendar)
OPCALC*     (Calculator)
OPDIR       (DirList, PathList)
OPEDIT      (LineEditor)
OPEDITOR    (TextEditor)
OPENTRY     (EntryScreen, ScrollingEntryScreen)
OPFORM      (PrintField, derived types, PrintedForm, Report, others)
OPHELP      (AbstractHelpWindow, PagedHelpWindow, ScrollingHelpWindow)
OPMEMO      (Memo, MemoFile)
OPMENU      (MenuItemNode, MenuItemList, SubMenu, Menu)
OPPICK      (PickList)
OPQKREF*    (QkRefChart)
OPSELECT    (AbstractSelector, Selector, ScrollingSelector)
OPWINDOW    (AbstractWindow, RawWindow, StackWindow, CommandWindow, others)
```

Of these units, the most important is OPWINDOW, which uses routines and objects in OPCRT, OPCMD, and OPFRAME to create the window objects from which most of the other objects in these units are derived. OPFORM uses OPABSFLD, OPPRNLOW, OPPRINT, and OPDEVICE as its primary building blocks. And OPEDIT builds upon OPABSFLD, OPFIELD, and OPFEDIT.

If you look at the object hierarchy in Appendix C, you'll see that most of the objects in these units *are not* descended from Root.

The units whose names are followed by asterisks (OPCAL, OPCALC, and OPQKREF) are bonus units that implement window-based objects. These units were written primarily for use in DESKPOP, but you may incorporate them into your own programs if you wish. Although these units are not documented in the manual, their objects are similar enough to all the other window-based objects that you should have little difficulty in figuring out how to use them.

B. Conventions and Guiding Principles

In writing **Object Professional**, we found it necessary to adopt a series of new programming conventions and philosophical principles, both to facilitate the development of the package and to make it easier for you to use the library. Most of these have to do primarily with objects, but others pertain equally well to units that contain only traditional procedures and functions.

The rest of this section outlines what these conventions and principles are and how they affect you. Knowing what they are is the first step toward seeing The Big Picture, the logic behind the hierarchy. The size and complexity of that hierarchy can be intimidating at first glance, but if you've read this section carefully you'll soon find that much of what you learn about a given object can be applied to others in the hierarchy as well.

Please be aware that in writing this section we have assumed that you are familiar with the basic concepts of object-oriented programming outlined in the Turbo Pascal 5.5 OOP Guide. There are no definitions of terms here, no tutorial on OOP. If you haven't read the OOP Guide, or felt hopelessly lost while reading it, you should definitely read (or reread) it before forging ahead here. On the other hand, if you are fairly comfortable with the concepts of inheritance, polymorphism, static and virtual methods, VMTs, constructors, and destructors, you should have no trouble here.

Static Methods vs. Virtual Methods

A sticky issue. In "pure" OOP languages, static methods don't exist; all methods are virtual. And an "OOP purist" would say that all the methods in our object hierarchy should be virtual, for the same reasons that virtual methods exist in the first place. There would be much truth to such an argument, but the larger truth is that making all methods virtual just isn't practical–at least not when you're programming with Turbo Pascal 5.5.

There are, we think, two good reasons for using static methods wherever possible. 1) Turbo Pascal's smart linker cannot strip unused virtual methods, only unused static methods; the mere act of instantiating an object of a given type causes all that object's virtual methods to be linked into the program. And 2) the more virtual methods an object has, the larger its VMT (Virtual Method Table) is: an object with 100 virtual methods would use over 400 bytes of space in the data segment. **Object Professional** doesn't have any objects with that many virtual methods, but it would have several if all methods were virtual.

Given that practical considerations make it impossible for all methods to be virtual, our philosophy is to make a method virtual only if 1) we know it must be overridden by a descendent object, 2) the method is intended to be overridden and exists for that purpose, or 3) the nature of the method might make it desirable to override it in a descendent object.

Procedure Pointers vs. Derived Types

Another philosophical issue. Consider the case of an object that needs to provide a hook that will allow the programmer to provide the object information that isn't necessarily known at compile time. The PickList object in OPPICK is a good example: it must provide a hook that allows you to tell it what the items in the pick list are.

The purist would say that the solution is to provide a dummy virtual method that is supposed to return the necessary information, then force the programmer to create a derived type that overrides the method. But this approach has two problems. First, it would be a nuisance to have to create a derived type *every* time you wanted to use a PickList (for example), especially when all you really need to write is a function to retrieve a string based on an item number. Second, it largely precludes the possibility of using the same PickList to display different lists.

Our preferred solution is a compromise. In such cases, there is indeed a virtual method that you can override if you want to. But there is also a means of specifying, via a procedure pointer, a user-written routine that does the job. The default implementation of the virtual method simply uses that procedure pointer to call your routine.

Constructors and Destructors

Object Professional follows Borland's convention of naming constructors "Init" and destructors "Done." Most objects also have a constructor called "Load" that is used for loading an object from a stream (more on streams in a moment)--another Borland convention.

For consistency's sake, every object in the hierarchy has a destructor called Done, even if it doesn't need one, and these destructors never take any parameters. With constructors, however, there is some degree of variation.

A constructor called Init always takes as few parameters as possible. If more parameters are desirable in certain situations, there is a second constructor called "InitCustom" that takes them (in addition to the ones passed to Init). In an object that has an InitCustom constructor, Init is typically implemented as a call to InitCustom that passes default values where necessary.

There are a few objects that have other constructors as well--MemoFile (OPMEMO) and PickList (OPPICK), for example--but in most of these cases their names start with "Init" and they take parameters similar to those for Init.

Options

Most of the larger objects in **Object Professional** provide a variety of on-off options that allow you to tailor their behavior to meet your preferences and needs. If you are familiar with Turbo Professional, you know that in the past we handled these options by means of global Boolean typed constants that you could modify or, in some cases, procedures that you could call to activate or deactivate the option. Both approaches have significant drawbacks. The global constant approach, besides wasting memory, can lead to problems when two objects are both active and each requires different options. The one-procedure-per-option approach can lead to an excessive number of procedure calls. So in **Object Professional** we have taken a different approach, one similar to that used in most C libraries.

All Boolean options are represented as bit flags. All objects that provide such options contain a field (a Byte, Word, or LongInt) that stores these bit flags, and a trio of methods for changing them or testing them: "xxOptionsOn" activates one or more options, "xxOptionsOff" deactivates one or more options, and "xxOptionsAreOn" returns True if the specified option(s) are on.

The "xx" is a one-, two- or three-character prefix that identifies the object. This prefix is needed because a given object may have more than one set of options. For example, the Memo object has a set of window-related options, inherited from RawWindow, as well as a set of memo-related options. To activate a window-related option, you would call wOptionsOn; to activate a memo-related option, you would call meOptionsOn.

These same prefixes are used in the names of the options, which are defined as constants. Here, for example, are the options for a Browser (OPBROWSE), which uses the "br" prefix:

```
brHexMode      = $01; {00000001 in binary}
brTabExpand    = $02; {00000010}
brStripHigh    = $04; {00000100}
brMousePage    = $08; {00001000}
brNewFile      = $10; {00010000}
brStreamReload = $20; {00100000}
```

As the comments on the right indicate, each of these values corresponds to a single bit: brHexMode to bit 0, brTabExpand to bit 1, and so on.

To activate a single option, you call xxOptionsOn with the name of the option. For example, to activate tab expansion in a Browser, you would call "brOptionsOn(brTabExpand)". To activate multiple options, you just add them together:
"brOptionsOn(brTabExpand+brStripHigh+brMousePage)".

In most cases, an object has a default set of options that is stored in a global typed constant with a name like "DefXxxOptions": for example, the default window options are stored in "DefWindowOptions." These options will be in effect unless you specifically deactivate them by calling xxOptionsOff. If necessary, you can change the default options by changing the typed constant.

In some cases, the desired option settings can or must be passed as one of the parameters to a constructor. All window-based objects, for example, have a constructor called InitCustom that takes a parameter specifying the window-related options for the object. For example:

```
MyWindow.InitCustom(1, 1, 80, 25, MyColorSet, wClear+wSaveContents)
```

If you called the Init constructor instead, the call would be simply

```
MyWindow.Init(1, 1, 80, 25)
```

Init would then call InitCustom, passing DefaultColorSet (which we'll discuss later) and DefWindowOptions as the last two parameters. If you want to use all the default window options *and* another option, you can OR the two together, like this:

```
MyWindow.InitCustom(..., DefWindowOptions or wAllMouseEvents)
```

Similarly, to use all the default options but one, you could mask it out using AND NOT:

```
MyWindow.InitCustom(..., DefWindowOptions and not wSaveContents)
```

It is not a good idea to use "+" instead of OR or "-" instead of AND NOT in these situations. "+" will work only if the option is known to be Off by default, and "-" will work only if the option is On by default. If the default options change, the meaning of your expression will be distorted.

Finally, be aware that many option settings are flags intended solely for internal use and must not be changed. When an object uses such flags, the unit that implements the object will have a global typed constant called "BadXxxOptions" that defines what those options are. The corresponding xxOptionsOn and xxOptionsOff methods use this information to prevent the options from being turned on or off. The main reason for making BadXxxOptions a typed constant (which can be changed) rather than a simple constant is to allow descendant objects that define new options to add to the list.

Naming Conventions

Wherever possible, we have tried to establish naming conventions that would make it easier to remember the names of methods and options. Several of these conventions have been mentioned already: constructors are usually called "Init," "InitCustom," or "InitXxx"; constructors that load an object from a stream are called "Load"; methods that store objects in a stream are called "Store"; destructors are called "Done"; option names and the methods that take options as parameters have a one- to three-character prefix that associates the name with the object. But there are several other naming conventions that you should be aware of.

1. Most objects have several methods, usually static, that are intended strictly for internal use. The names of these methods are always preceded by the same one- to three-character prefix used for option names.
2. This prefix is also used for the names of fields in the object. This convention allows you to tell, based on the name of a field, what object in the hierarchy defined it.
3. This prefix is also used for virtual methods that you should not call yourself, but you might want to override in a descendent type. (See the discussion of "Procedure Pointers vs. Derived Types," above.)
4. There are a number of objects types in the library that exist only to be overridden, objects that you would never want to create an instance of. Such objects usually have a name that starts with the word "Abstract."
5. The name of an object's command processor is always "XxxCommands," where "Xxx" identifies the object it goes with.
6. The name of the stream registration routine for an object is always "XxxStream," where "Xxx" identifies the object it goes with.

Streams

With a few minor exceptions, all of the objects in the library are designed to allow them to be stored in streams. Any object that can be stored in a stream has a constructor called Load and a corresponding method called Store. For a detailed discussion of streams, see the documentation for OPROOT.

Heap Usage

All of the objects in the library use the heap to store data in cases where the size of the data can vary, as well as in cases where the number of data items can vary (e.g., the linked list of fields in a data entry screen and the list of items in a menu). This practice leads to efficient use of available memory, but it also has one potentially significant drawback: it can lead to heap fragmentation problems in certain cases.

Programs that create and destroy lots of objects in no particular order (DESKPOP is a prime example), and more especially programs that allow the user to zoom and resize windows, are prime candidates for problems with heap fragmentation. We devoted considerable thought to this issue, hoping to find a way to avoid these problems altogether, but came to the conclusion that all the possible solutions were either too clumsy, too intrusive, or too risky. The only reasonable solution to this problem, we think, is for Borland to provide a heap manager that can perform garbage collection more or less transparently.

On a less gloomy and more general note: The OPROOT unit, which is used by most of the other units containing objects, installs a heap error function that causes all failed calls to New or GetMem to return Nil. If necessary, you can override this behavior by installing your own heap error function, but be sure that you understand the possible consequences before doing so. If at all possible, we recommend that you write your program so that it works under these conditions. One thing that you can do to make your job easier is to use the GetMemCheck and FreeMemCheck functions in OPROOT whenever you need to allocate or deallocate memory from the heap; this is what all of our objects do. See the documentation for OPROOT for details on the use of these functions.

Error Handling

All of the higher level objects in the library are designed to report all errors that occur after the object has been instantiated to a central error handler. This error handler is passed both an error code and (usually) a default error message, allowing the error handler to display a generic error message or a customized one. Errors that occur in constructors (before an error handler

can be specified) are stored in a global variable declared in the OPROOT unit, InitStatus. For details concerning the way errors are handled, see the documentation for the CommandWindow object (OPWINDOW) and Appendix D ("Error Codes").

Command and Keyboard Handling

All of the objects in the library that act on keyboard commands use an object called a CommandProcessor to translate keystrokes into commands. To make this possible, the object provides the CommandProcessor with a command table, an array of a specific format that defines the relationships between key sequences and commands. Each command in the library is assigned a unique code. The use of command codes and command tables make it possible not only for a program to modify and add key assignments at runtime, but also to write separate configuration programs that allow the user to modify the assignments. For details concerning the handling of commands, see the documentation for OPCMD.

The CommandProcessor also provides several programming hooks, the most notable of which allows you to specify a routine to be called when a keystroke needs to be obtained from the user. This hook makes it possible to perform "background tasks" while waiting for keyboard input. This subject is also discussed in the documentation for OPCMD.

Mouse Support

All of the objects in the library that use a CommandProcessor provide integral mouse support, and all objects based on the CommandWindow provide specific support for scroll bars and mouse hot spots. For details concerning the way that mouse support is implemented, see the documentation for the OPMOUSE, OPCMD (CommandProcessor object), OPFRAME (ScrollBar and HotSpot objects), and OPWINDOW (CommandWindow object) units.

Context-Sensitive Help Hooks

All of the objects in the library that use a CommandProcessor also provide programming hooks that facilitate the development of a context-sensitive help system. For details concerning the use of these hooks, which are all largely identical, see the documentation for the OPCMD (CommandProcessor object), OPWINDOW (CommandWindow object), and OPHELP units.

Colors and ColorSets

In **Object Professional**, all of objects that perform screen I/O are designed to make it easy to write programs that adapt automatically to the differences between color and monochrome systems. When an object needs you to specify a video attribute, it therefore asks for two attributes rather than one: one for color systems, one for monochrome systems. The object will later use a simple routine in the OPCRT unit to choose the appropriate video attribute in a given situation.

These objects are also designed to allow your color choices to be centralized in a single data structure, an object called the ColorSet, that can be reused throughout the program. When you instantiate an object that uses lots of different video attributes, you simply pass it the name of your ColorSet. For a discussion of the ColorSet object, and further details on what it is used for, see the documentation for OPCRT.

Configuration Data and ICD Files

In order to facilitate the development of configuration programs–programs that modify other programs, as DPINST does–all units that contain typed constants that a configuration program might need to modify have an include file with an extension of ICD (Include Configuration Data). In most cases, these files simply contain command tables. The purpose of these files is to make it possible for a configuration program to include copies of these tables, and other

configuration data, without using the units they go with. For an example of how ICD files can be used, see DPINST.PAS.

Conditional Compilation Directives

Obviously, in a library as large as **Object Professional**, there will be many, many features of most units that you will not need in a given program. In most cases, these "features" consist of procedures, functions, and methods that can be eliminated by the smart linker when your program is compiled. In other cases, however, the mere existence of these features causes code to be linked in, making the program larger than it needs to be. To address this problem, we have defined a series of conditional compilation directives that can disable these features and prevent the corresponding code from being linked in.

There are also cases where a particular feature does not cause a significant amount of code to be linked in, but it produces behavior that is undesirable in certain situations. Conditional compilation directives are also used to address this issue in cases where no better solution is available.

All of these compilation directives may be found in the file OPDEFINE.INC, which is included in every unit in the library. In some cases the directive is defined there; in others, it is present in the file but disabled by placing a period in front of the $DEFINE directive, like so:

```
{.$DEFINE SomeDirective}
```

To enable a directive that is not defined by default, load OPDEFINE.INC into an editor and remove the period in front of the "$". To disable a directive that is defined, insert a period. In either case, you will need to recompile the entire library after editing OPDEFINE.INC. The easiest way to do this is to use the supplied OPRO.MAK file in conjunction with Borland's MAKE utility. See the "Installation" section of the Introduction (Chapter 1) for details.

The following is an alphabetical list of all the conditional compilation directives in OPDEFINE.INC. In each case except the first, the first line of the entry shows the name of the directive, and the second and following lines indicate the default state of the directive ("Defined" or "Undefined") and give a brief description of what it affects.

$N-
This directive determines whether the Float type in OPSTRING is defined as an Extended ($N+) or a Real ($N-). It also controls the inclusion of certain methods in various units that apply only to 8087 numeric types.

FastDispose
Defined. This directive should not be defined if the caller of OPSORT needs to perform heap allocation or deallocation while the sort is in progress--that is, within the user-defined procedures of the sort. For large sorts with element size exceeding 4 bytes, FastDispose makes a significant difference in the speed of heap deallocation when the sort is finished. See the documentation for OPSORT for details.

FourByteDates
Defined. If this directive is defined, dates in OPDATE are stored as Longints, giving a range of 01/01/1600-12/31/3999. If it is not defined, dates are stored as Words, giving a range of 01/01/1900-12/31/2078. See the documentation for OPDATE for details.

InitAllUnits
Defined. This directive forces all units to contain an initialization block, even those that do not have any actual initialization code. Adding these blocks helps to work around a bug in Turbo

Debugger 1.5 that makes the debugger unable to locate symbol information for certain units if one of the units in the program has had all its code stripped by the smart linker. Borland has a simple patch to correct this bug, and if you have applied the patch you will want to undefine InitAllUnits and recompile all the units in the library.

PickListFields
Defined. This directive enables support for pick list fields in OPENTRY. Note that some of the demo programs cannot be compiled if this directive is not defined.

ThwartSideKick
Defined. If this directive is defined, OPTSR and OPSWAP try to thwart SideKick. That is, they attempt to bypass the code in SideKick's INT $8 (clock tick) handler that makes it unsafe to load other TSRs after SideKick. See the documentation for OPTSR for details.

UseAdjustableWindows
Defined. This directive enables the AdjustWindow, ResizeWindow, MoveWindow, and certain other methods in OPWINDOW. Note that some of the demo programs cannot be compiled if this directive is not defined.

UseBCD
Undefined. This directive enables support for editing of BCD variables in OPEDIT and OPENTRY, and for printing them with OPFORM.

UseCrt
Undefined. This directive determines whether or not OPCRT is compiled in such a way as to allow it to coexist peacefully with the standard Turbo Pascal CRT unit. Ordinarily, defining UseCrt does nothing but waste code space. It is useful only in cases where your program contains a unit that USES CRT and you do not have the source code for the unit. See the documentation for OPCRT for details.

UseDates
Defined. This directive enables support for editing of date and time variables in OPEDIT and OPENTRY, and for printing them with OPFORM. It also enables certain options in OPDIR. Note that some of the demo programs cannot be compiled if this directive is not defined.

UseHotSpots
Defined. This directive enables hot spot support in OPWINDOW and all units that depend on it. Note that some of the demo programs cannot be compiled if this directive is not defined.

UseMouse
Defined. This directive enables mouse support in all of the units that provide it, as well as in most of the demo programs.

UseScrollBars
Defined. This directive enables scroll bar support in OPWINDOW and all units that depend on it. Note that some of the demo programs cannot be compiled if this directive is not defined.

UseShadows
Defined. This directive enables support for shadows in OPWINDOW. Note that some of the demo programs cannot be compiled if this directive is not defined.

3. Screen and Keyboard

A. Overview

OPCRT is a plug-in replacement for Turbo Pascal's standard CRT unit. Although completely compatible with CRT where it counts, OPCRT offers an impressive array of extensions and enhancements. You'll find routines to move the cursor and change its shape; read characters and strings from the screen; write directly to video memory at top speed; save and restore windows; switch in and out of the EGA's 43-line mode and the VGA's 50-line mode without clearing the screen; poll the keyboard; the list goes on and on. OPCRT also allows you to write to any video page on a color adapter (even with WriteLn), and it supports multitaskers such as DesqView and OmniView.

OPCOLOR simply defines a series of constants such as 'WhiteOnBlack' and 'BlackOnLtGray' that can be used to refer to particular video attributes.

OPMOUSE offers a comprehensive set of low-level routines that simplify the task of adding mouse support to your programs. You'll find numerous routines for reading the mouse buttons, reading the position of the mouse cursor, hiding and unhiding the mouse cursor, and so on. These routines form the basis for the high-level mouse support provided in OPWINDOW, OPENTRY, OPMENU, and other units that supply window-based objects.

Finally, OPCMD provides facilities for translating keystrokes and mouse button presses into abstract "commands" (move cursor up, move cursor down, quit, etc.). In addition to defining a standard set of command codes, OPCMD defines two object types: the CommandProcessor, which is used to translate keys into commands by scanning through user-defined command tables, and the CommandPacker, which is used to unpack the data stored in a command table, making it easier to modify. CommandProcessors are used throughout **Object Professional** to insure that key assignments can be easily modified. The CommandPacker object facilitates the writing of installation programs such as DPINST.

B. OPCRT: Enhanced CRT Unit

As indicated at the start of the chapter, OPCRT is a much enhanced version of Turbo's CRT unit. If you don't use any of its extensions, your programs will almost always behave just as they would if you used CRT instead. There are a few subtle but important differences, however:

1. OPCRT supports all video pages on color adapters, not just page 0. If you switch video pages by calling SetVisiblePage, you can continue to use Read, Write, GotoXY, and so on. CRT won't let you do that.
2. Unlike CRT, OPCRT is very much dependent on certain global variables it defines: CurrentPage, CurrentMode, ScreenWidth, ScreenHeight, CurrentDisplay, and VideoSegment. Whereas CRT is constantly calculating the starting address for video memory, for example, OPCRT assumes that the value in VideoSegment is correct. Making such assumptions saves time, and allows numerous routines to work with virtual screens that otherwise could not, but it also means that in general you must use OPCRT's routines to change video modes, change display pages, or switch in/out of 43-line/50-line mode. If you use another routine to perform such a task (a routine in a graphics package, for example), you need to then call either GetCrtMode (reinitializes CurrentMode, ScreenWidth, ScreenHeight, CurrentPage, CurrentDisplay, VideoSegment) or ReinitCrt (also reinitializes CheckSnow and the current window coordinates). Also, if you are writing a TSR that uses OPCRT, you need to call one of these two routines on entry to your popup, usually ReinitCrt.
3. All keyboard input routines are written with TSRs in mind. If you were to write a TSR that used the regular CRT unit's input routines (ReadKey, Read, ReadLn), you would frequently lock out other TSRs: SideKick, for example, would not be able to pop up on top of your TSR while you were waiting for a key. OPCRT avoids this problem.
4. OPCRT restores the global variable BufLen, available in Turbo 3 but gone in later versions of Turbo Pascal, which allows you to restrict the length of strings read with Read or ReadLn statements. For example, if you needed to restrict MyString to 10 characters:

   ```
   BufLen := 10;
   ReadLn(MyString);
   ```

   BufLen defaults to 126, and it is reinitialized to that value after each Read or ReadLn.
5. The initialization code for OPCRT automatically recognizes Compaqs and Zeniths (not all models, just some), whose display adapters do not require snow prevention, and sets the CheckSnow flag accordingly. This allows for dramatically improved screen performance on these machines.
6. OPCRT supports multitaskers such as DesqView and OmniView.

The remainder of this section is divided into two parts. Part 1 covers issues related to the numerous procedures and functions found in OPCRT. Part 2 covers the one object implemented in OPCRT, the ColorSet.

1. Procedures and Functions

Compatibility with CRT

Although highly compatible with CRT, OPCRT is not interchangeable with CRT as far as the standard units GRAPH3 and TURBO3 are concerned. Both USE CRT, and conflicts will arise if you try to USE either of them in a program that also USES OPCRT, since both CRT and OPCRT would then be present in the same program. To prevent the two units from fighting for control of the screen, OPCRT contains code that checks for the presence of CRT and aborts the program if it is found. An initial check is done when the program starts, and further checks are done later when certain key procedures are called (in case CRT is initialized after OPCRT).

The best solution to this problem is to not use GRAPH3 or TURBO3. These units are provided by Borland to make it easier to convert programs written with earlier versions of the compiler. Once a program has been converted, you would be well advised to remove all dependencies on these units by using the GRAPH unit instead of GRAPH3, and by eliminating the calls to routines in TURBO3.

If this solution is unacceptable, however, there is an alternative. In OPDEFINE.INC, you will find the following statement:

```
{.$DEFINE UseCrt}
```

If you remove the '.' in front of '$' and recompile OPCRT, it will then be completely compatible with CRT. All calls to routines in CRT will be redirected to the equivalent routines in OPCRT, and all global variables that are declared in both units will be shared in common. That is, the variable in OPCRT will be declared absolute on top of the variable of the same name in CRT: for example, 'TextAttr : Byte absolute Crt.TextAttr'. This workaround has several disadvantages, mainly in terms of wasted code space, but it is an acceptable solution in cases where there are no better alternatives. (Special thanks to Lane Ferris and to the folks at Software Science, Inc., formerly known as The Research Group, for their help in developing this workaround.)

The same issue arises, of course, if you try to use OPCRT together with a non-standard unit that USES CRT. This is less of a problem than with GRAPH3 and TURBO3, since you will typically have the source code for the unit, and can simply replace CRT with OPCRT in the USES statement. But you may encounter difficulties when using pre-compiled units distributed in TPU form either as shareware or as commercial software. Here again, you may solve the problem by defining the UseCrt directive in OPDEFINE.INC.

Experience has shown that the unit that gives the most people problems in this regard is the TACCESS unit in the Turbo Database Toolbox. You can correct the problem simply by changing the USES statement in TACCESS.PAS. In cases where you don't know what units you're using USE CRT, you may want to use Borland's GREP utility to search all your source files for instances of "CRT". We recommend using the following command line:

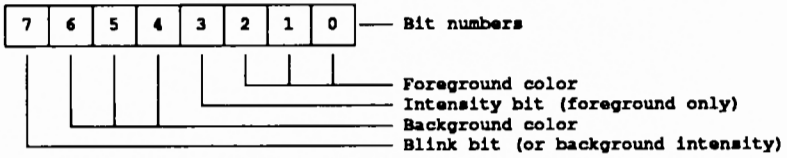
```
GREP -r- -w+ "CRT" *.pas
```

The "-w+" option is particularly useful, as it will rule out most instances of "OPCRT".

Video Attributes

One concept that you absolutely must understand in order to use OPCRT and the units that depend on it is that of the video attribute. When running in text mode, the video memory on a PC is divided into a series of rows (usually 25) and columns (usually 80). For each of these

locations on the screen, there are two bytes of data: one contains the character at that location, and the other contains its color characteristics. These characteristics are described in a bit-mapped byte known as a video attribute, which has four components:



The formula for calculating a video attribute's value, then, is:

```
(BackgroundColor * 16) + ForegroundColor
```

To calculate the video attribute for Yellow on Red, for example, you would multiply the background color, Red (= 4), by 16 and add the foreground color, Yellow (= 14):

```
(Red * 16) + Yellow = (4 * 16) + 14 = 64 + 14 = 78 = $4E
```

Here are some more examples of various color combinations and their corresponding video attributes:

Foreground	Background	Blink	Binary	Hex	Decimal
Light gray (07)	Black (00)	No	00000111	07	7
White (15)	Black (00)	No	00001111	0F	15
Yellow (14)	Blue (01)	No	00011110	1E	30
Light green (10)	Red (04)	No	01001010	4A	74
White (15)	Brown (06)	Yes	11101111	EF	239

If you don't want to take the time to learn how to calculate video attributes in your head, or if you just want to save time, we recommend that you keep DESKPOP loaded while you are programming. The quick reference chart in DESKPOP will let you find the proper value quickly just by scrolling through the chart until you find the color combination you want. Alternatively, you can use the color constants declared in the OPCOLOR unit, discussed in the next section. It allows you to use mnemonic names like 'WhiteOnBlack' and 'BlackOnLtGray' to refer to particular video attributes.

One further note on video attributes. In the diagram above, you'll notice that bit 7 of the attribute is labelled "Blink bit (or background intensity)". Normally, any video attribute with a value greater than 127 (01111111 in binary, or $7F in hex) will produce blinking text. However, if you disable attribute blinking using the SetBlink routine in OPCRT, the "blink bit" will be interpreted by the display adapter as signifying the intensity of the background color. To see the effects of turning blinking off, activate the quick reference chart in DESKPOP, press <CtrlPgDn> to go to the end of the chart, then press <CtrlB> to toggle attribute blinking. As you'll see, the number of non-blinking color combinations will be doubled, from 128 to 256.

Row,Col vs. X,Y coordinates

Another point you must understand in order to use OPCRT is that most of its routines ask that screen locations be expressed in terms of Rows and Columns, rather than X,Y (Column,Row) coordinates. The reasons for this are partly historical, partly religious. There are some exceptions, most notably among routines found in both CRT and OPCRT, but the following rules generally apply:

1. Any routine that requires you to specify the position of the cursor (such as GotoXY) assumes X,Y notation.

2. Any routine that requires you to specify the location where text is to be displayed (such as FastWrite) assumes Row,Column notation.
3. Any routine that requires you to specify window coordinates (such as Window) assumes XLow, YLow, XHigh, YHigh notation.

If you can remember those three rules, you should have no difficulty remembering which notation to use for any given routine.

Enhanced keyboard support

Unlike CRT, OPCRT is capable of supporting enhanced keyboards, meaning that its routines for reading from the keyboard (ReadKey, ReadKeyWord, Keypressed, etc.) are capable of recognizing keys such as <F11> and <F12> that are not found on the older PC- and AT-style keyboards. By default, this support is disabled, but you can easily enable it by adding the following statement to the beginning of your program:

```
UseEnhancedKbd := EnhancedKbdInstalled;
```

UseEnhancedKbd is a global variable that affects the behavior of the various keyboard routines. If it is False, these routines use standard BIOS function calls that work on all machines, with any kind of keyboard. If it is True, they instead use special BIOS functions that work only when an enhanced keyboard is installed.

The EnhancedKbdInstalled function attempts to determine whether or not an enhanced keyboard is installed. The test it uses is not foolproof. It is guaranteed to work on PS/2s, and as far as we know it will never report that an enhanced keyboard is installed when there isn't one, but it does sometimes fail to detect that an enhanced keyboard is installed when there is. To deal with these situations, we provide a tiny program called EKFIX.COM that you may distribute with your own programs if you wish. This utility simply toggles a single bit of a byte in low memory, a bit that is supposed to be set if an enhanced keyboard is installed (bit 4 of the byte at $40:$96). The message displayed by the program indicates whether the bit flag was turned on (enabled) or off (disabled).

Note: Before using the UseEnhancedKbd hook to enable enhanced keyboard support in OPCRT, we strongly recommend that you read the documentation for the OPENHKBD unit. In many cases, using OPENHKBD is a better choice, since it allows older keyboards to emulate enhanced keyboards to a great extent.

Declarations found in both CRT and OPCRT

In general, the following constants, variables, and routines are functionally identical to those in Turbo's standard CRT unit. Only a few special points are worth noting about them:

1. Although OPCRT's TextMode routine reinitializes some important global variables (the same ones that GetCrtMode initializes), it does not initialize all of them. Therefore, after switching video modes, you should call ReinitCrt to insure that they are all initialized properly. OPCRT's TextMode also differs from the standard version in that it allows you to switch into graphics modes with it. (Note that support for graphics in OPCRT in general is severely limited, however.)
2. DirectVideo has the same range of influence that it does in Turbo's CRT unit. That is, it determines how Write/WriteLn, Read/ReadLn, and GotoXY display text on the screen and move the cursor. It does not, however, affect the behavior of OPCRT's non-standard screen writing routines, such as FastWrite and FlexWrite, which always write directly to video memory.

3. ClrScr can fill a window or screen using a character other than a space. For example, to fill the screen with asterisks, you would set TextChar to '*' before calling ClrScr. TextChar is automatically reset to ' ' after the screen has been cleared.

Constants

```
Black = 0;                {color constants}
Blue = 1;
Green = 2;
Cyan = 3;
Red = 4;
Magenta = 5;
Brown = 6;
LightGray = 7;
DarkGray = 8;
LightBlue = 9;
LightGreen = 10;
LightCyan = 11;
LightRed = 12;
LightMagenta = 13;
Yellow = 14;
White = 15;
Blink = 128;
BW40 = 0;                 {video mode constants}
CO40 = 1;
C40 = CO40;
BW80 = 2;
CO80 = 3;
C80 = C80;
Mono = 7;
Font8x8 = 256;
```

Variables

```
CheckBreak : Boolean;     {enables Ctrl-Break checking}
CheckEOF : Boolean;       {enables Ctrl-Z checking}
CheckSnow : Boolean;      {set to True to prevent snow on CGAs}
DirectVideo : Boolean;    {write directly to screen?}
LastMode : Word;          {current video mode in low byte, 8x8 flag in high}
SaveInt1B : Pointer;      {previous INT $1B handler}
TextAttr : Byte;          {current video attribute}
WindMin : Word;           {Window XLow and YLow: in 0..24, 0..79 format}
WindMax : Word;           {Window XHigh and YHigh: in 0..24, 0..79 format}
```

Procedures

```
procedure AssignCrt(var F : Text);
  {-Routes standard screen I/O through TPCRT rather than DOS}

procedure ClrEol;
  {-Clears the remainder of the current screen line}

procedure ClrScr;
  {-Clears the current window}

procedure Delay(MS : Word);
  {-Delay for MS milliseconds}

procedure DelLine;
  {-Deletes current screen line}

procedure GoToXY(X, Y : Byte);
  {-Moves cursor to column X, row Y, relative to Window}
```

```
procedure HighVideo;
  {-Sets intensity bit in TextAttr}
procedure InsLine;
  {-Inserts a new line at the position of the cursor}
procedure LowVideo;
  {-Clears intensity bit in TextAttr}
function KeyPressed : Boolean;
  {-Returns True if a key has been pressed}
procedure NormVideo;
  {-Sets TextAttr to video attribute used at start of program (no longer
    equivalent to HighVideo!)}
procedure NoSound;
  {-Turns off the sound}
function ReadKey : Char;
  {-Reads a character from the keyboard}
procedure Sound(Hz : Word);
  {-Turns on the sound at the designated frequency}
procedure TextBackground(Color : Byte);
  {-Sets background color for screen writes}
procedure TextColor(Color : Byte);
  {-Sets foreground color for screen writes}
procedure TextMode(Mode : Word);
  {-Sets current video mode}
function WhereX : Byte;
  {-Returns current X (column) coordinate of cursor, relative to Window}
function WhereY : Byte;
  {-Returns current Y (row) coordinate of cursor, relative to Window}
procedure Window(XLow, YLow, XHigh, YHigh : Byte);
  {-Sets current window coordinates}
```

Types

```
ColorChoice = (UseDefault, ForceColor, ForceMono);
```

Type of DefColorChoice, which influences the result returned by the UseColor function (called whenever a choice between color and monochrome attributes must be made). The normal setting is UseDefault, which means that the choice is based on the current video mode and/or the display type. ForceColor forces color attributes to be chosen; ForceMono forces monochrome attributes to be chosen.

```
ColorMapTable = array[0..255] of Byte;
```

A table used to map color video attributes for monochrome or black and white displays.

```
CursorType = (cuNormal, cuFat, cuBlock, cuHidden, cuUnknown);
```

Standard cursor shapes defined by OPCRT. cuNormal is the thin cursor, normally two scan lines high, that is the default on PCs and compatibles. cuFat is a slightly fatter cursor, usually three lines high. cuBlock is a "block cursor," a full screen line (8-14 scan lines) in height. cuHidden is a hidden cursor. cuUnknown is used internally to classify "non-standard" cursor shapes.

```
DisplayType = (MonoHerc, CGA, MCGA, EGA, VGA, PGC);
```

The six standard types of display adapters recognized by OPCRT. MonoHerc means either a plain monochrome adapter or a Hercules monochrome graphics card. PGC stands for IBM Professional Graphics controller card.

```
FlexAttrs = array[0..3] of Byte;
```

An array of attributes for use by the FlexWrite procedure.

```
FrameCharType = frTL..frRR;
FrameArray    = array[FrameCharType] of Char;
```

An array of characters used for drawing frames around windows.

```
HercCardType = (HercNone, HercPlain, HercPlus, HercInColor);
```

Used to classify the various types of Hercules graphics cards. HercPlain indicates a standard Hercules Graphics Card or compatible. HercPlus indicates a Hercules Graphics Card Plus (with RamFont capabilities). HercInColor indicates a Hercules InColor Card. HercNone is any card that does not fit into one of the other categories.

```
WindowCoordinates =
  record
    XL, YL, XH, YH : Byte;
  end;
```

A record used to save/restore window coordinates.

Constants

```
AutoMapMono = $FF;
```

When passed as the second parameter to the MapMono routine, or any routine that calls MapMono, this value indicates that the monochrome attribute should be selected automatically using a translation table (ColorMap). See the discussion of ColorSets in the next section.

```
BiosScroll : Boolean = True;
```

If False, a special scrolling routine is used that suppresses snow and flicker on color graphics adapters. This setting affects the behavior not only of Write and WriteLn statements, but also ScrollWindowUp and ScrollWindowDown. The BIOS scrolling routine is usually faster on CGAs, but far less appealing to the eye.

```
ColorMap : ColorMapTable = (...);
```

A translation table used to convert color video attributes into monochrome video attributes. This table is used by MapColor, MapMono, and the PackedWindow.MapColors method.

```
DefaultPasswordChar = ^G;
```

The default character used to represent text when a line/field editor is in "password mode." Used by OPEDIT and OPENTRY; not used by OPCRT itself.

```
DefaultPadChar = ' ';
```

The default character used to pad strings (visually) being edited with a line or field editor. Used by OPEDIT and OPENTRY; not used by OPCRT itself.

```
DefaultColorSet : ColorSet = (...);
```

The default colors for all screen-oriented objects. See the discussion of ColorSets in the next section.

```
DefColorChoice : ColorChoice = UseDefault;
```

Influences the behavior of the UseColor function. See the entry for the ColorChoice type, above, as well as the entry for UseColor.

```
DetectMultitasking : Boolean = False;
```

If DetectMultitasking is True, OPCRT will perform the checks necessary to determine if the program is being run under a multitasking system such as DesqView or OmniView (formerly TaskView). If it is, MultitaskingOn will be set to True, and all screen writes (using Write/WriteLn, FastWrite, etc.) will be directed to the virtual screen supplied by the system rather than to the physical screen. Since the default setting for DetectMultitasking is False, you need to add two lines to the start of your program if you want it to be able to run as a background task:

```
DetectMultitasking := True;
ReinitCrt;
```

This feature allows most programs written with OPCRT to be run as background tasks inside a window (the only exceptions we know of are programs that use alternate video pages). They can also be considered "TopView aware" under the definition used by DesqView and OmniView. Note that OPCRT does not support TopView itself, nor does it support Microsoft Windows or Double DOS.

Important note: Do not set DetectMultitasking to True in a TSR. The function call used to determine if a multitasker is in use can cause problems if it is executed inside a partition that has been designated as "writing directly to screen." This is why DetectMultitasking defaults to False. If you know that you won't be writing TSRs, you can change the default to True and recompile OPCRT.

```
DisplayOverride : ShortInt = -1;
```

Because auto-detection of display adapters is an imperfect art, this variable is provided as a means of overriding it. It is initialized to -1, which enables auto-detection. If you need to force OPCRT to treat, say, an EGA as a CGA, you can set DisplayOverride to the ordinal value of the appropriate DisplayType--"DisplayOverride := Ord(CGA);"--then call ReinitCrt. The more common use for this variable is to force OPCRT to recognize an adapter as an EGA or VGA in cases where auto-detection has failed to identify it correctly. This should rarely be necessary, however.

```
frTL = 0;
frBL = 1;
frTR = 2;
frBR = 3;
frTT = 4;
frBB = 5;
frLL = 6;
frRR = 7;
```

These constants represent the component parts of a FrameArray: frTL = top left corner, frBL = bottom left corner, frTR = top right corner, frBR = bottom right corner, frTT = top edge, frBB = bottom edge, frLL = left edge, frRR = right edge.

```
FrameChars : FrameArray = 'ÉÈ»¼Í=ºº';
```

The default characters used to draw frames around windows.

```
HercExceptionReg : Byte = $27;
```

This variable is a hook that makes it possible to use OPCRT as the basis for a more complete set of routines for programming the newer Hercules graphics cards. It is needed because the Exception Register on a RamFont card is a write-only register. HercExceptionReg retains the value of the last byte written to this port, allowing you (in theory) to independently program the three functions controlled by the Exception Register without affecting the other bits.

```
InsertCursor   : CursorType = cuFat;
OvertypeCursor : CursorType = cuNormal;
CtrlCharCursor : CursorType = cuBlock;
```

These three constants are used by OPSEDIT, OPFEDIT, and OPMEMO to determine the shape of the cursor while editing in insert and overtype modes, and when waiting for a control character to be entered after the ccCtrlChar command has been issued. The default settings call for a thin cursor in overtype mode, a slightly fatter cursor in insert mode, and a block cursor while waiting for a control character.

```
TextChar : Char = ' ';
```

Together with TextAttr, this character is used by ClrScr, ClrEol, ScrollWindowUp, and ScrollWindowDown when clearing or scrolling a window. Much like BufLen, it is automatically reset to its default value of ' ' after the screen has been cleared/scrolled.

```
UseEnhancedKbd : Boolean = False;
```

If this variable is set to True, all keyboard calls in OPCRT (ReadKey, ReadKeyWord, KeyPressed, etc.) will use the BIOS's special enhanced keyboard functions, allowing them to recognize the keys found only on enhanced keyboards, such as <F11> and <F12>. See the

section on "Enhanced keyboard support," above, and the entry for EnhancedKbdInstalled, below, for details.

Variables

```
BufLen : Word;
```

Maximum length of a string for Read/ReadLn. The valid range is 1..126, and BufLen is always reset to 126 after each Read operation.

```
CtrlBreakFlag : Boolean;
```

Set to True when <CtrlBreak> has been pressed. For internal use only.

```
CurrentDisplay : DisplayType;
```

The currently selected display adapter. If two display adapters are installed, CurrentDisplay tells you only the type of the adapter currently in use. See also the entry for EnhancedDisplay.

```
CurrentMode : Byte absolute LastMode;
```

The current video mode.

```
CurrentPage : Byte;
```

The current video page.

```
EnhancedDisplay : DisplayType;
```

This variable is meaningful only if it is set to MCGA, VGA, or EGA. It is useful primarily in detecting situations where two adapters are installed, a plain monochrome adapter and an EGA attached to a color monitor, for example. In such a case EnhancedDisplay would be set to EGA, even though CurrentDisplay was set to MonoHerc. This information is especially needed by memory resident programs that need to support the 43- and 50-line modes (which use the 8x8 font) available on the EGA and VGA, respectively. See SMACS.PAS for an example. If an MCGA, VGA, or EGA is not installed, EnhancedDisplay will be set to MonoHerc.

```
InTextMode : Boolean;
```

Set to False when in graphics mode. (Note that Hercules graphics mode is not detected.)

```
MultiTaskingOn : Boolean;
```

This variable is set to True if DetectMultitasking is True and a multitasker (DesqView, TaskView/OmniView) was detected.

```
NormalAttr : Byte;
```

The video attribute for NormVideo.

```
OneMS : Word;
```

This variable contains the loop count used by Delay to generate a 1 ms delay.

```
SaveCtrlC : Boolean;
```

Used to save the state of DOS's break checking option. OPCRT forces break checking off when a program begins, and restores it to its former state when a program ends. See the Turbo Pascal manual's discussion of the GetCBreak and SetCBreak routines for details.

```
ScreenHeight : Word;
```

The current height of the display (usually 25).

```
ScreenWidth : Word;
```

The current width of the display.

```
VideoSegment : Word;
```

The segment portion of the current starting address for video memory. Normally this will be $B000 if a monochrome adapter is in use, or $B800 if a color adapter is in use, but other values are possible if CurrentPage < > 0.

```
VirtualHeight : Word;
```

This variable is normally equal to ScreenHeight, but it is sometimes changed by OPWINDOW to allow certain screen writing routines to be used on virtual screens. See the documentation for OPWINDOW.

```
VirtualSegment : Word;
```

This variable is normally equal to VideoSegment, but it is sometimes changed by OPWINDOW to allow FastWrite and other screen writing routines to be used on virtual screens. See the documentation for OPWINDOW.

```
VirtualWidth : Word;
```

This variable is normally equal to ScreenWidth, but it is sometimes changed by OPWINDOW to allow FastWrite and other screen writing routines to be used on virtual screens. See the documentation for OPWINDOW.

```
WhichHerc : HercCardType;
```

Indicates the type of Hercules graphics card that is installed and active, if any. It is initialized when the program begins, and it is reinitialized each time that ReinitCrt is called. WhichHerc will be set to HercNone if a Hercules card is not installed or not active.

BlockCursor / ChangeAttribute

Declaration

```
procedure BlockCursor;
```

Purpose

Set scan lines for a block cursor.

Description

This routine changes the cursor scan lines such that the cursor is as large as possible. On machines with a CGA installed, this means that the cursor will use all 8 scan lines; on machines with a monochrome adapter, EGA, or VGA, it will use all 14 scan lines.

See Also

FatCursor
HiddenCursor
NormalCursor
SetCursorSize

Declaration

```
procedure ChangeAttribute(Number : Word; Row, Col : Word; Attr : Byte);
```

Purpose

Change Number video attributes to Attr starting at Row,Col.

Description

This routine allows you to alter screen colors without altering the text on the screen. A typical application for ChangeAttribute might be in a moving bar menu, since it allows you to highlight (or "unhighlight") a particular region of the screen without knowing what it is that's being highlighted. See the demonstration program SMACS.PAS for an example of this technique.

Number specifies the number of video attributes to be changed. Row and Col indicate the starting screen coordinates for the region to be changed. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Attr specifies the new video attribute for the region.

Example

```
program Colors;
uses OPCRT;
const Attr : Byte = 0;
begin
  ClrScr;
  WriteLn('Hello multi-colored world!'^M^J'Now watch the colors change...');
  Delay(1000);
  while not KeyPressed do begin
    ChangeAttribute(26, 1, 1, Attr and $7F); {suppress blink bit}
    Delay(50); {make it a little easier on the eyes...}
    Inc(Attr); {change the attribute}
  end;
end.
```

Cycles rapidly through all color combinations until a key is pressed.

Declaration

```
function CheckKbd(var KeyCode : Word) : Boolean;
```

Purpose

Returns True (and the key codes) if a keystroke is waiting.

Description

CheckKbd is similar to KeyPressed, but if a key is waiting it will also return the key code (in the same form as ReadKeyWord). It thus has the advantage of letting you see what the pending key is, without having to call ReadKey or ReadKeyWord, both of which will remove the keystroke from the BIOS's keystroke buffer.

Example

```
var KeyCode : Word;
...
if CheckKbd(KeyCode) then
  if Lo(KeyCode) = 27 then begin  {check for <Esc>}
    KeyCode := ReadKeyWord;       {'eat' the keystroke}
    Exit;                         {and exit}
  end;
```

Check for a pending keystroke; if <Esc> was pressed, 'eat' the keystroke and exit, otherwise continue.

See Also

ReadKeyWord

Declaration

```
function ClassifyCursorType : CursorType;
```

Purpose

Classify cursor type based on scan lines.

Description

This function looks at the shape of the cursor and classifies it based on OPCRT's standard cursor types.

Example

```
CT := ClassifyCursorType;
SelectFont8x8(not Font8x8Selected);
SetCursorType(CT);
```

Save the cursor type, toggle between 25- and 43-/50-line mode, then restore the previous cursor type. (SelectFont8x8 always resets the cursor to cuNormal.)

See Also

SetCursorType

ClearPage / ClearWindow / ColorMono

Declaration

```
procedure ClearPage(PageNo : Byte);
```

Purpose

Clear the specified video page with TextChar and TextAttr.

Description

This routine clears the specified video page with the character in TextChar (normally ' ') and the video attribute in TextAttr. This routine should never be used on monochrome adapters, which have only one video page. It is intended to be used to clear inactive video pages before calls to SetActivePage.

Example

See the example for SetActivePage.

See Also

SetActivePage SetVisiblePage

Declaration

```
procedure ClearWindow(XLow, YLow, XHigh, YHigh : Word; Ch : Char; A : Byte);
```

Purpose

Clear a window using the specified character and attribute.

Description

This routine clears the window defined by XLow, YLow, XHigh, and YHigh using the specified character (Ch) and video attribute (A). Unlike ScrollWindowUp and ScrollWindowDown, this routine will work with virtual screens as well as the physical screen.

Example

```
ClearWindow(1, 2, 80, ScreenHeight-1, #176, $07);
```

Fill all lines except the top and bottom ones with a polka-dot block character.

See Also

ScrollWindowDown ScrollWindowUp

Declaration

```
function ColorMono(Color, Mono : Byte) : Byte;
```

Purpose

Choose between a color and a monochrome video attribute.

Description

This function is one of the most widely used routines in **Object Professional**. It is called any time one of the screen-oriented objects has to choose between two video attributes, Color and Mono. A simple routine, it returns the Color video attribute if the UseColor function returns True, or False if it doesn't. For consistency, you should use it in your own display routines whenever you have to make a similar choice.

Example

```
FastWrite('Hello World!', 1, 1, ColorMono($1F, $0F));
```

Displays 'Hello World!' in White on Blue on color displays, White on Black on monochrome or black and white displays.

See Also

UseColor

Declaration
```
function CursorEndLine : Byte;
```

Purpose
Returns the ending scan line of the cursor.

Description
This routine returns the ending scan line for the cursor, normally either 7 (in a color text mode) or 13 (in a monochrome text mode).

Example
See below.

Declaration
```
function CursorStartLine : Byte;
```

Purpose
Returns the starting scan line of the cursor.

Description
This routine returns the starting scan line for the cursor, normally either 6 (in a color text mode) or 12 (in a monochrome text mode). In the EGA's 43-line mode, the starting scan line is typically 5.

Example
```
NormalCursor; {select normal two-line cursor}
WriteLn('Starting scan line for cursor: ', CursorStartLine);
WriteLn('Ending scan line for cursor: ', CursorEndLine);
```
This would display 5, 6, or 12 for the starting scan line, depending on the current mode, and 7 or 13 for the ending scan line

---

Declaration
```
function CursorTypeSL : Word;
```

Purpose
Returns a word containing the starting and ending scan lines of the cursor.

Description
This routine combines the functions of CursorEndLine and CursorStartLine by returning both values. The starting scan line is returned in the high byte of the word, the ending line in the low byte. It can sometimes be used, in combination with SetCursorSize, as a substitute for GetCursorState and RestoreCursorState.

Example
```
var CursorScanLines, Key : Word;
begin
  CursorScanLines := CursorTypeSL;
  HiddenCursor;
  FastWrite('Error. Press any key to continue...', 25, 1, TextAttr);
  Key := ReadKeyWord;
  SetCursorSize(Hi(CursorScanLines), Lo(CursorScanLines));
end;
```
Save the cursor size, hide the cursor, display an error message at the bottom of the screen, wait for a keystroke, then restore the cursor size.

See Also
GetCursorState SetCursorSize

EnhancedKbdInstalled / FastCenter

Declaration

```
function EnhancedKbdInstalled : Boolean;
```

Purpose

Returns True if an enhanced keyboard is installed.

Description

This routine attempts to determine whether or not an enhanced keyboard is installed. The test it uses, a simple one, is identical to the one used by the OPENHKBD unit. For a complete discussion of the method used, see the section on "Enhanced keyboard support" at the beginning of the chapter.

The typical use for this function is to initialize the UseEnhancedKbd variable, which determines whether or not keyboard routines such as ReadKey, ReadKeyWord, and KeyPressed use the BIOS's special enhanced keyboard function calls. If UseEnhancedKbd is False, as it is by default, the regular BIOS functions are used, and key combinations not found on "regular" keyboards will be ignored (e.g., <F11>, <^Up>, <AltTab>).

The one disadvantage to enabling enhanced keyboard support is that it will sometimes prevent older TSRs from popping up in situations where it is safe for them to do so.

Example

```
UseEnhancedKbd := EnhancedKbdInstalled;
if UseEnhancedKbd then begin
  Write('Press the <F12> key...');
  repeat
  until ReadKeyWord = $8500; {wait for <F12> to be pressed}
end
else
  WriteLn('Enhanced keyboard not installed');
```

If an enhanced keyboard is installed, the repeat loop would be executed until <F12> was pressed.

Declaration

```
procedure FastCenter(St : string; Row, Attr : Byte);
```

Purpose

Write St centered on window Row in Attr.

Description

This routine displays the specified string (St) centered within the current window on the specified Row in the designated video attribute (Attr). Row is assumed to be relative to the current window.

Example

```
Window(2, 2, 79, 24);
FastCenter('Hello world!', 1, $F);
```

This would display 'Hello world!' on row 2 (absolute), centered between columns 2 and 79.

See Also

FastFlush

Declaration
```
procedure FastFill(Number : Word; Ch : Char; Row, Col : Word; Attr : Byte);
```

Purpose
Fill Number video words with Ch/Attr starting at Row,Col.

Description
This routine will fill a contiguous region of video memory starting at Row,Col with Number instances of the specified character/attribute pair (Ch and Attr). A typical use of the routine might be to fill an entire row of the screen with a line-drawing character, such as #196.

Example
```
FastFill(ScreenWidth, #196, 13, 1, $F);
```
Fill all of row 13 with character #196, displayed in white on black.

Declaration
```
procedure FastFlush(St : string; Row, Attr : Byte);
```

Purpose
Write St flush right on window Row in Attr.

Description
This routine displays the specified string (St) flush right against the edge of the current window on the specified Row in the designated video attribute (Attr). Row is assumed to be relative to the current window.

Example
```
Window(2, 2, 79, 24);
FastFlush('Hello world!', 1, $F);
```
This would display 'Hello world!' on row 2 (absolute), with the end of the string ('!') touching the edge of the window at column 79.

See Also
FastCenter

Declaration
```
procedure FastRead(Number : Byte; Row, Col : Word; var St : string);
```

Purpose
Read Number characters from the screen into St starting at Row,Col.

Description
Like ReadCharAtCursor, FastRead reads text off the screen, but it does so far more rapidly. It is most useful, perhaps, in popups written with the OPTSR unit, which might want to inspect the contents of the screen being displayed by the program that was interrupted.

Number is the number of characters to read off the screen. Row and Col specify the location to start reading from. (Note that these screen coordinates are treated as absolute, not relative to the current window.) And St is a string variable in which the characters read off the screen are returned.

See SMACS.PAS for an example of a practical application for FastRead.

See Also
ReadCharAtCursor

FastText / FastVert

Declaration

```
procedure FastText(St : string; Row, Col : Word);
```

Purpose

Write St at Row,Col without changing the underlying video attribute.

Description

This routine is identical to FastWrite except that it takes no attribute parameter. Text is displayed in whatever colors happen to be on the screen at the location where the string is written.

Example

```
TextAttr := $F;
ClrScr;
FastText('This text will be displayed in white on black', 1, 1);
```

Display the specified string at the top left corner of the screen in whatever colors are already present on row 1.

See Also

FastWrite

Declaration

```
procedure FastVert(St : string; Row, Col : Word; Attr : Byte);
```

Purpose

Write St vertically at Row,Col in Attr (video attribute).

Description

This routine is identical to FastWrite except that it writes text vertically rather than horizontally.

Example

```
FastWrite('Writing horizontally...', 1, 2, TextAttr);
FastVert('Writing vertically...', 2, 1, TextAttr);
```

The first string would be written horizontally across the top of the screen, and the second would be written vertically along the left edge of the screen.

See Also

FastWrite

Declaration

```
procedure FastWrite(St : string; Row, Col : Word; Attr : Byte);
```

Purpose

Write St at Row,Col in Attr.

Description

This routine offers a convenient means of displaying text on the screen at top speed without causing snow or flicker. Unlike a standard Write statement, FastWrite in itself gives you complete control over where a string is displayed and in what video attribute, eliminating the need for a cumbersome sequence like

```
GotoXY(column, row);
TextColor(n);
TextBackground(n);
Write(s);
```

The only significant limitation of FastWrite is that it can be used only in text modes.

St is the string to be displayed. Row and Col specify where the string is to be displayed. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Attr is the video attribute in which it is to be displayed.

Example

```
var
  Row, I : Byte;
begin
  ClrScr;
  Write('Press any key for a demonstration of fast screen writing...');
  I := ReadKeyWord;
  while not KeyPressed do
    for Row := 1 to ScreenHeight do
      for I := 0 to 7 do
        FastWrite('FastWrite ', Row, Succ(I*10), Lo(Random(127)));
  I := ReadKeyWord;
end;
```

Declaration

```
procedure FastWriteAttr(St : String; Row, Col : Word; AttrSt : String);
```

Purpose

Write St at Row,Col using attributes in AttrSt.

Description

FastWriteAttr is similar to FastWrite, except that the final parameter is a string of video attributes. Each element of AttrSt corresponds to a single character in St: St[1] will be displayed in the video attribute in AttrSt[1], St[2] in AttrSt[2], and so on. AttrSt is assumed to be of the same length as St.

This routine is typically used in conjunction with ReadAttribute, which returns a string of video attributes, but it may also be used instead of FlexWrite in cases where even greater flexibility is required.

Example

```
const ASt : string[4] = #$1F#$2F#$3F#$4F;
...
FastWriteAttr('TEST', 1, 1, ASt);
```

Displays each character in 'TEST' in a different color (blue, green, cyan, and red).

See Also

FlexWrite ReadAttribute

FastWriteCtrl / FatCursor

Declaration

```
procedure FastWriteCtrl(St : String; Row, Col : Word; Attr, Ctrl : Byte);
```

Purpose

Write St at Row,Col in Attr (video attribute) without snow.

Description

This routine is identical to FastWrite except that it displays control characters as uppercase letters in the video attribute specified by Ctrl. Non-control characters are displayed normally in Attr.

Example

```
FastWriteCtrl('Control characters: '^A^B^C^D, 1, 1, $07, $0F);
```

Displays the first part of the string normally in light gray on black; the final four characters (^A^B^C^D) as uppercase letters in white on black.

See Also

FastWrite

Declaration

```
procedure FatCursor;
```

Purpose

Set larger scan lines for cursor based on current video mode.

Description

This routine changes the cursor scan lines such that the cursor is slightly larger (fatter) than the normal two-line cursor. On machines with a color adapter installed, the cursor will use 3 of the 8 available scan lines; on machines with a monochrome adapter, it will use 4 of the 14 available scan lines; and in the EGA's 43-line or the VGA's 50-line mode it will use 5 of the 8 available scan lines.

See Also

BlockCursor
HiddenCursor
NormalCursor
SetCursorSize

Declaration

```
function FlexLen(S : string) : Byte;
```

Purpose

Return actual (visible) length of a flex string.

Description

This function takes a string that is intended to be passed to FlexWrite and calculates its actual length (the physical length of the string minus the number of special control characters in it). It was written primarily for internal use by OPPICK, but you may use it if you need it.

Example

```
FlexLen('This '^A'is a '^B'test')
```

Returns 14. That is, Length('This is a test').

Declaration

```
procedure FlexWrite(St : string; Row, Col : Word; var FAttrs : FlexAttrs);
```

Purpose

Write St at Row,Col with flexible color handling.

Description

This routine is similar to FastWrite, but the last parameter, instead of being a single byte that indicates the video attribute for the string, is an array[0..3] of byte. The purpose of the array is to allow different portions of the string to be displayed in different attributes, and the signal used to indicate a change of attributes is a control character: ^A, ^B, ^C.

The default attribute for the string is FAttrs[0]. To change attributes, you would insert a ^A (for FAttrs[1]), ^B (FAttrs[2]), or ^C (FAttrs[3]). To return to the previous attribute, you would insert the same control character a second time. See the examples below.

Note that each time you switch to a new attribute, 2 bytes of stack space are used up. (It is recovered when you restore the previous attribute, or when the routine ends, whichever comes first). It is up to the programmer to insure that adequate stack space is available, although this is rarely a concern.

See PTIME.PAS and OPCALC.PAS for further examples of the use of FlexWrite.

Examples

```
const FAttrs : FlexAttrs = ($7, $F, $E, $70);
FlexWrite('This '^A'is'^A' '^B'a'^B' '^C'test', 1, 1, FAttrs);
```

In this example, each of the four words in the string is displayed in a different attribute. The first word and the spaces between words are all displayed in the default attribute (FAttrs[0]).

```
FlexWrite('This '^A'is also a '^B'test', 1, 1, FAttrs);
```

The first word is displayed in FAttrs[0], the next three in FAttrs[1], and the last word in FAttrs[2].

See Also

FastWrite FastWriteAttr

Font8x8Selected / FrameWindow

Declaration

```
function Font8x8Selected : Boolean;
```

Purpose

Return True if EGA or VGA is active and using the 8x8 font.

Description

This routine returns True if an EGA display adapter is active and in 43-line mode, or a VGA adapter is active and in 50-line mode. Calling Font8x8Selected also has the effect of reinitializing the global variable ScreenHeight.

Note: OPCRT's initialization code automatically senses 43-line/50-line mode and sets the default window coordinates to 1,1,80,43/50 if it is found. If your application depends on having a 25-line display, you should add the following code to the beginning of your program:

```
Window(1,1,80,25); {force a 25-line window}
if Font8x8Selected then
  SelectFont8x8(False);
```

Example

```
if CurrentDisplay >= Ega then
  SelectFont8x8(not Font8x8Selected);
```

Toggles an EGA from 25- to 43-line (or a VGA from 25- to 50-line) mode or vice versa.

See Also

SelectFont8x8

Declaration

```
procedure FrameWindow(LeftCol, TopRow, RightCol, BotRow : Word;
                      FAttr, HAttr : Byte; Header : string);
```

Purpose

Draws a frame around a window.

Description

This routine draws a frame (or box) around a window whose dimensions are defined by LeftCol, TopRow, RightCol, and BotRow. (Note that these screen coordinates are treated as absolute, not relative to the current window.) The characters used to draw frames are stored in the global typed constant FrameChars. Header is a string to be displayed at the center of the top row of the window. If no header is desired, you can pass an empty string. FAttr is the video attribute to be used in displaying the frame characters, and HAttr is the video attribute for the Header.

Example

```
FrameWindow(1, 1, ScreenWidth, ScreenHeight, TextAttr, TextAttr,
            ' Header Goes Here ');
```

Draws a frame around the entire screen.

Declaration

```
function GetCrtMode : Byte;
```

Purpose

Get the current video mode.

Description

Besides returning the current video mode, GetCrtMode reinitializes key global variables used by OPCRT: CurrentMode (current video mode), InTextMode (False when in graphics mode), ScreenWidth (current width of the screen), CurrentPage (current display page), and VideoSegment (segment portion of starting address for video memory). It is intended primarily to be used in popup routines as a fast alternative to ReinitCrt, which reinitializes several other global variables and takes longer to execute.

For a good example of a case where GetCrtMode should be used, see PTIME.PAS. PTIME calls it before displaying its on-screen clock, to make certain that (1) the clock is displayed at the top right hand corner of the screen, (2) the display is not in graphics mode, and (3) the clock is displayed on the current video page of the active display adapter.

See Also

Font8x8Selected ReinitCrt

Declaration

```
procedure GetCursorState(var XY, ScanLines : Word);
```

Purpose

Return the current position and size of the cursor.

Description

This routine, when used in combination with RestoreCursorState, provides a convenient means of saving and later restoring the current size and position of the cursor. It is useful not only in popup routines written with OPTSR, but also in 'black box' routines that need to be able to manipulate the size of the cursor without affecting the calling program's display.

Example

```
var XY, CursorScanLines, Key : Word;
begin
  GetCursorState(XY, CursorScanLines);
  HiddenCursor;
  FastWrite('Error. Press any key to continue...', 25, 1, TextAttr);
  Key := ReadKeyWord;
  RestoreCursorState(XY, CursorScanLines);
end;
```

See any of the memory resident demonstration programs for further examples.

See Also

RestoreCursorState

GetPagePointer / GetPageSegment / GotoXYAbs

Declaration

```
function GetPagePointer(PageNo : Byte) : Pointer;
```

Purpose

Get a pointer to the start of the specified video page.

Description

Given a valid video page number, this function returns a pointer to the start of the video memory for that page. The result it returns is equivalent to the expression 'Ptr(GetPageSegment(PageNo), 0)'.

Declaration

```
function GetPageSegment(PageNo : Byte) : Word;
```

Purpose

Get the video segment corresponding to the specified video page.

Description

Given a valid video page number, this function returns the segment portion of the corresponding address in video memory. If PageNo is invalid, GetPageSegment returns VideoSegment.

GetPageSegment is intended primarily for internal use by SetActivePage and GetPagePointer, but you may also want to use it in conjunction with VirtScreen.Init. That is, if your program runs only on color displays, you might want to use alternate video pages as virtual screen buffers to conserve memory. See the documentation for VirtScreen.Init.

See Also

VirtScreen.Init

Declaration

```
procedure GotoXYAbs(X, Y : Byte);
```

Purpose

Move cursor to column X, row Y. No error checking done.

Description

Like GotoXY, GotoXYAbs moves the cursor to a specific spot on the screen, but the coordinates passed are treated as absolute, rather than relative to the current window. All coordinates are considered 1-based (top left corner is 1,1), and no range checking is done (the cursor can be moved outside the current window).

Example

```
Window(3,3,80,25);
GotoXYAbs(3,3);
```

Moves cursor to the top left corner of the window, just as 'GotoXY(1,1)' would.

See Also

WhereXAbs
WhereYAbs
WhereXY

Declaration
```
function HercGraphicsMode : Boolean;
```

Purpose
Return True if a Hercules card is in graphics mode.

Description
This routine returns True if a Hercules card (or close compatible) is installed and in graphics mode. Be forewarned that the test used here is far from perfect. It is guaranteed to work for true Hercules cards, and it works with most of the better Hercules clone cards, but there also many cases where it will fail. The most common cases are EGA/VGA clone cards that provide Hercules emulation. To determine whether or not this routine will be reliable, call HercModeTestWorks. Note that it is not possible to determine which graphics page (0 or 1) is active.

Example
```
if HercPresent then begin
  if HercModeTestWorks and HercGraphicsMode then
    SetHercMode(False, 0);
  ...
end;
```
This checks to see if a Hercules card is installed and in graphics mode. If it is, the display is forced back into text mode.

See Also
HercPresent SetHercMode

Declaration
```
function HercModeTestWorks : Boolean;
```

Purpose
Return True if HercGraphicsMode will work.

Description
This routine is intended to be used in conjunction with the HercGraphicsMode routine. Its purpose is simply to determine whether or not the test used by that routine is reliable.

HercPresent / HiddenCursor

Declaration

```
function HercPresent : Boolean;
```

Purpose

Return True if a Hercules graphics card is present.

Description

This routine returns True if a Hercules graphics card is installed in the system. Note, however, that this does not necessarily mean that the current display is a Hercules card: on a dual-display system, the Hercules might be the inactive display.

Example

```
if HercPresent then begin
  WriteLn('This program cannot be used on a Hercules display adapter');
  Halt;
end;
```

Halt the program if a Hercules display adapter is installed.

Declaration

```
procedure HiddenCursor;
```

Purpose

Hide the cursor.

Description

This routine hides the hardware cursor on any display adapter. Note, however, that the cursor can sometimes be mysteriously "unhidden" on systems with monochrome adapters installed when the display is scrolled by the BIOS. This is a quirk of the hardware, and it is generally unavoidable.

See Also

BlockCursor
FatCursor
NormalCursor
SetCursorSize

Declaration

```
function KbdFlags : Byte;
```

Purpose

Returns BIOS's keyboard status flags.

Description

This routine returns the current value of the BIOS's keyboard status flags, a bit-coded byte located at $40:$17. Although this value can be obtained directly by using an absolute variable, the KbdFlags uses the more reliable method of executing BIOS interrupt $16, function 2. The meaning of the individual bits in the flags byte are shown below.

Bit	Val	Meaning
7	$80	Insert on
6	$40	Caps Lock on
5	$20	Num Lock on
4	$10	Scroll Lock on
3	$08	<Alt> depressed
2	$04	<Ctrl> depressed
1	$02	<LeftShift> depressed
0	$01	<RightShift> depressed

Example

```
var
  KbdFlagsByte : Byte;
begin
  KbdFlagsByte := KbdFlags;
  if (KbdFlagsByte and $10) <> 0 then
    WriteLn('Scroll Lock on');
end;
```

Declaration

```
function MakeHiddenAttr(A : Byte) : Byte;
```

Purpose

Make foreground and background colors the same.

Description

This inline macro was written primarily for internal use by OPENTRY and other units that hide text by displaying it in a video attribute that has the same foreground and background colors. You may use it anytime you need to do the same thing in your own code.

Examples

```
MakeHiddenAttr($07) {light gray on black}
```

Returns $00 (black on black)

```
MakeHiddenAttr($1F) {white on blue}
```

Returns $11 (blue on blue)

MapColor / MapMono

Declaration

```
function MapColor(C : Byte) : Byte;
```

Purpose

Map a video attribute for visibility on Mono/BW displays.

Description

If the UseColor function returns True, this function simply returns the input parameter (C) unchanged. Otherwise, MapColor will attempt to map C to a visually appealing value on a display without color by using it as an index into the ColorMap array. A perfect mapping is unfortunately impossible, but MapColor does a good job in most cases. Note that ColorMap is a global typed constant, so you can change the way that colors are mapped simply by changing the array.

Example

```
FastWrite('hello', 1, 1, MapColor($1C));
```

On a color system, this will display a red 'hello' on a blue background. On a monochrome or black and white display, the string will be dim white on a black background. Try running the WINWOW demonstration program in color and black-and-white (or monochrome) modes to get a feel for what MapColor does.

See Also

UseColor

Declaration

```
function MapMono(Color, Mono : Byte) : Byte;
```

Purpose

Return a mapped color video attribute if Mono = $FF.

Description

Given a color and a monochrome video attribute, this function will return either Mono or, if Mono = $FF, ColorMap[Color]. This function is intended primary for internal use.

Examples

```
MapMono($4F, $07)
```

Returns $07.

```
MapMono($4F, $FF)
```

Returns ColorMap[$4F] ($70 by default).

Declaration

```
procedure MoveScreen(var Source, Dest; Length : Word);
```

Purpose

Move Length words from Source to Dest without snow.

Description

MoveScreen is similar to Turbo's standard Move procedure, but it is intended to be used strictly for moving data between normal memory and video memory, or from one region of video memory to another. It also differs from the standard Move routine in that it cannot move single bytes of data, only pairs of bytes (words).

Source is the address to move data from, and Dest the address to move data to. If the segment portion of either address is in the region of memory used by a CGA ($B8xx) and CheckSnow is True, MoveScreen will inhibit snow and flicker; otherwise, it will move the data as rapidly as possible. Length is the number of words to be moved.

Example

```
var
  ScreenBufPtr : Pointer;
  BytesOnScreen : Word;
begin
  BytesOnScreen := (ScreenWidth * ScreenHeight) * 2;
  GetMem(ScreenBufPtr, BytesOnScreen);
  MoveScreen(Mem[VideoSegment:0], ScreenBufPtr^, BytesOnScreen div 2);
  {-- do something --}
  MoveScreen(ScreenBufPtr^, Mem[VideoSegment:0], BytesOnScreen div 2);
  FreeMem(ScreenBufPtr);
end;
```

Saves the screen and later restores it. Note that this particular task can be accomplished more easily by using SaveWindow and RestoreWindow.

See Also

RestoreWindow SaveWindow

Declaration

```
procedure NormalCursor;
```

Purpose

Set normal scan lines for cursor based on current video mode.

Description

This routine restores the cursor to its normal size–usually two scan lines high.

See Also

BlockCursor HiddenCursor
FatCursor SetCursorSize

ReadAttrAtCursor / ReadAttribute

Declaration

```
function ReadAttrAtCursor : Byte;
```

Purpose

Returns attribute at the current cursor location on the selected page.

Description

This routine is similar to ReadCharAtCursor, except that instead of returning the character beneath the cursor it returns the video attribute that the character is displayed in.

Example

```
CharAtCursor := ReadCharAtCursor;
AttrAtCursor := ReadAttrAtCursor;
```

Save the character and attribute beneath the cursor in CharAtCursor and AttrAtCursor, respectively.

See Also

ReadCharAtCursor

Declaration

```
procedure ReadAttribute(Number : Byte; Row, Col : Word; var St : string);
```

Purpose

Read Number attributes from the screen into St starting at Row,Col.

Description

This routine is similar to the FastRead routine except that instead of reading characters off the screen it reads video attributes. It is also similar to ReadAttrAtCursor, but it is much faster.

Number is the number of video attributes to read off the screen. Row and Col specify the location to start reading from. And St is a string variable in which the attributes read off the screen are returned.

Example

```
ReadAttribute(ScreenWidth, 1, 1, S);
for I := 1 to ScreenWidth do
  Byte(S[I]) := SwapNibble(Byte(S[I])) and $7F;
WriteAttribute(S, 1, 1);
```

Reverses all the foreground and background colors on row 1.

See Also

FastRead
FastWriteAttr
WriteAttribute

Declaration

```
function ReadCharAtCursor : Char;
```

Purpose

Returns character at the current position of the cursor.

Description

Like FastRead, ReadCharAtCursor can be used to read data off the screen. Although ReadCharAtCursor reads only a single character at a time, and requires that the cursor be positioned properly, it has the advantage of working equally well in the standard CGA graphics modes (assuming that simple Write and WriteLn statements have been used to display text). FastRead can be used only in text modes.

Example

```
var X, Y, XY : Word;
...
XY := WhereXY;
for Y := 1 to ScreenHeight do begin
  for X := 1 to ScreenWidth do begin
    GotoXY(X, Y); Write(Lst, ReadCharAtCursor);
  end;
  WriteLn(Lst);
end;
GotoXYAbs(Lo(XY), Hi(XY));
```

This code does the same thing as the BIOS's print screen routine.

Declaration

```
function ReadKeyWord : Word;
```

Purpose

Waits for keypress, then returns scan and character codes together.

Description

Unlike ReadKey, this routine returns both the scan code and the character code for the key that was pressed. The low byte of the word returned contains the character code, and the high byte contains the scan code.

ReadKeyWord has two advantages over ReadKey. First, it allows you to get all the information you need with a single function call. Second, it lets you distinguish between keys that share the same character code but different scan codes. For example, on a standard PC or AT keyboard, there are two keys that return '+' (#43): one on the top row of the main cluster of keys, and one on the numeric keypad. The first one has a scan code of 13 ($0D), and the second (sometimes referred to as the 'gray plus key') has a scan code of 78 ($4E). Using ReadKeyWord, you can easily distinguish the two if desired, as is shown below.

Example

```
var Key : Word;
    Ch : Char absolute Key;
...
Key := ReadKeyWord;
if Ch = '+' then
  if (Hi(Key) = $4E) then
    WriteLn('Gray plus key pressed')
  else
    WriteLn('White plus key pressed');
```

See Also

CheckKbd

ReInitCrt / RestoreCursorState / RestoreWindow

Declaration

```
procedure ReInitCrt;
```

Purpose

Reinitialize OPCRT unit's internal variables.

Description

ReinitCrt is intended primarily for use in 1) TSRs, 2) programs that use ExecDos or OPEXEC, and 3) programs that use graphics. It completely reinitializes all of the mode-sensitive global variables used by OPCRT: CurrentMode (current video mode), InTextMode (False when in graphics mode), ScreenWidth (current width of the screen), ScreenHeight (current height of the screen), WindMin and WindMax (window dimensions–the entire screen is made the current window), CurrentPage (current video page), CurrentDisplay (the type of the current display adapter), CheckSnow (inhibit snow on color displays), and VideoSegment (segment portion of the starting address for video memory).

In general, there are only three situations in which you should need to call ReinitCrt: on entry to a popup, after executing another program, or after switching into or out of graphics mode.

See Also

ExecDos (OPDOS) GetCrtMode

Declaration

```
procedure RestoreCursorState(XY, ScanLines : Word);
```

Purpose

Reset the cursor to a position and size saved with GetCursorState.

Description

This routine, when used in combination with GetCursorState, provides a convenient means of saving and later restoring the current size and position of the cursor. It is particularly useful not only in popup routines written with the OPTSR unit, but also in 'black box' routines that need to be able to manipulate the size of the cursor without having the change affect the calling program's display.

Example

See the entry for GetCursorState.

Declaration

```
procedure RestoreWindow(XLow, YLow, XHigh, YHigh : Byte;
                        Deallocate : Boolean; var Covers : Pointer);
```

Purpose

Restore screen contents and deallocate buffer space if requested.

Description

When used in conjunction with SaveWindow, this routine provides a convenient means of saving and restoring all or part of the screen (a window).

The dimensions of the window to be restored are defined by XLow, YLow, XHigh, and YHigh. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Although the window coordinates will ordinarily be the same as those passed to SaveWindow, they don't have to be. Covers is a pointer to the buffer being used to store the screen data. Deallocate indicates whether the buffer should be deallocated after the window is restored.

Example

See the entry for SaveWindow.

Declaration

```
procedure RestoreWindowCoordinates(WC : WindowCoordinates);
```

Purpose

Restore previously saved window coordinates.

Description

This routine allows you to restore window coordinates previously saved using SaveWindowCoordinates. See the entry for StoreWindowCoordinates for an example.

See Also

StoreWindowCoordinates

Declaration

```
function SaveWindow(XLow, YLow, XHigh, YHigh : Byte; Allocate : Boolean;
                    var Covers : Pointer) : Boolean;
```

Purpose

Allocate buffer space if requested and save window contents.

Description

When used in conjunction with RestoreWindow, this routine provides a convenient means of saving and restoring all or part of the screen (a window). The dimensions of the window to be saved are defined by XLow, YLow, XHigh, and YHigh. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Allocate indicates whether or not a buffer should be allocated for the screen data. If Allocate is True, a pointer to the buffer is returned in Covers. Otherwise, Covers must be a pointer to a buffer large enough to store the data. SaveWindow will return False only if Allocate is True and insufficient memory remains on the heap to allow a buffer to be allocated.

Since SaveWindow uses MoveScreen to move data, snow and flicker are inhibited if necessary.

Note: SaveWindow and RestoreWindow are intended to be used only while in text modes, not graphics modes.

Examples

```
var
 BufPtr : Pointer;
begin
 if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, True, BufPtr) then
   Exit;
 {--do something--}
 RestoreWindow(1, 1, ScreenWidth, ScreenHeight, True, BufPtr);
end;
```

Correct. This would allocate a buffer and save the entire screen, later restoring it with RestoreWindow.

```
if not SaveWindow(1, 1, ScreenWidth, ScreenHeight, True, BufPtr^) then
  Exit;
```

Incorrect! The '^' in 'BufPtr^' is not needed.

See Also

MoveScreen RestoreWindow

ScrollWindowDown / ScrollWindowUp

Declaration

```
procedure ScrollWindowDown(XLo, YLo, XHi, YHi, Lines : Byte);
```

Purpose

Scrolls the designated window down the specified number of lines.

Description

This routine provides an alternative to InsLine in cases where you don't want to move the cursor, or you need to scroll a region of the screen other than the current window.

The window to be scrolled is defined by XLo, YLo, XHi, and YHi. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Lines is the number of lines to be scrolled. If Lines = 0, the entire window is cleared.

Like InsLine, ScrollWindowDown fills the blank lines it creates with spaces, which are displayed in the current video attribute (TextAttr).

Note that, if BiosScroll is false, a special scrolling routine is used that inhibits snow and flicker on color graphics adapters.

Example

```
var
  I : Word;
begin
  ClrScr;
  for I := 1 to ((ScreenHeight-2) * ScreenWidth) do
    Write('*');
  FrameWindow(1, 1, ScreenWidth, ScreenHeight-2, TextAttr, TextAttr,
    ' ScrollWindowDown Demonstration ');
  Delay(2000);
  ScrollWindowDown(2, 2, Pred(ScreenWidth), ScreenHeight-3, 1);
end;
```

Fills all but the last two lines of the screen with asterisks, draws a frame around the region, then scrolls the window inside the frame down one line.

Declaration

```
procedure ScrollWindowUp(XLo, YLo, XHi, YHi, Lines : Byte);
```

Purpose

Scrolls the designated window up the specified number of lines.

Description

This routine provides an alternative to DelLine in cases where you don't want to move the cursor, or you need to scroll a region of the screen other than the current window.

The window to be scrolled is defined by XLo, YLo, XHi, and YHi. (Note that these screen coordinates are treated as absolute, not relative to the current window.) Lines is the number of lines to be scrolled. If Lines = 0, the entire window is cleared.

Like DelLine, ScrollWindowUp fills the blank lines it creates with spaces, which are displayed in the current video attribute (TextAttr).

Note that, if BiosScroll is false, a special scrolling routine is used that inhibits snow and flicker on color graphics adapters.

Example

See the example for ScrollWindowDown, above.

Declaration

```
procedure SelectFont8x8(On : Boolean);
```

Purpose

Toggle 8x8 font (43-line mode or 50-line mode) on or off.

Description

This routine allows you to switch in (On = True) and out (On = False) of the EGA's 43-line mode and the VGA's 50-line mode. If an EGA or VGA is not installed, it does nothing. After the switch into 43-line/50-line mode has been made, the global variable ScreenHeight will be set to 43 or 50 rather than its usual value of 25. The current window coordinates, however, will not be changed, since doing so is not always desirable. Changing them to reflect the new screen dimensions is the responsibility of the programmer.

This routine does not clear the screen, as does the only method offered by the standard CRT unit:

```
TextMode(CurrentMode+Font8x8);
```

Note: Due to a flaw in the EGA BIOS, a normal two-line cursor (with a starting scan line of 6, and an ending scan line of 7) will be hidden after switching to 43-line mode if CGA cursor emulation is left on. To prevent this, SelectFont8x8 automatically turns emulation off when switching into 43-line mode, and turns it on again when switching back into 25-line mode.

Example

```
if CurrentDisplay < Ega then
  WriteLn('EGA or VGA required')
else begin
  Write('This is 25-line mode... ');
  Delay(1000);
  SelectFont8x8(True);
  WriteLn('and this is ', ScreenHeight, '-line mode');
  Delay(1000);
  SelectFont8x8(False);
end;
```

See Also

Font8x8Selected

SetActivePage / SetBlink

Declaration

```
procedure SetActivePage(PageNo : Byte);
```

Purpose

Selects the video page that will be written to with subsequent operations on the screen.

Description

This routine allows you to direct screen output to an alternate video page. A typical application for it might be in a case where you need to draw a complicated screen, and you don't want the user to be able to see it until it's finished. See the example below.

SetActivePage affects FastWrite, FlexWrite, and all other screen writing routines in OPCRT that do not use BIOS calls (as ReadCharAtCursor does, for example). It does not, however, affect Write/Ln or Read/Ln.

Note that this function does *not* do the same thing as Graph.SetActivePage.

Example

```
ClearPage(1);
SetActivePage(1);
{...write to page 1...}
SetVisiblePage(1);
{...do something...}
SetVisiblePage(0);
```

Clear page 1, redraw the screen without letting the user see the work in progress, make it visible, and then switch back to page 0.

See Also

ClearPage SetVisiblePage

Declaration

```
procedure SetBlink(On : Boolean);
```

Purpose

Enable text mode attribute blinking if On is True.

Description

SetBlink enables or disables blinking text. If it is off, "bright" colors such as pink ($D) and yellow ($E) may be used as background colors.

To see what effects can be achieved with SetBlink(Off), load the demonstration DESKPOP, open the quick reference chart window, then press <CtrlPgDn> to go to the end of the chart, then <CtrlB> to turn blinking off. All values greater than 127 will then correspond to colors with a brighter background.

Example

```
TextAttr := $8F;
WriteLn('This produces blinking text');
SetBlink(False);
WriteLn('This doesn''t, even though the attribute is the same');
```

The first statement is displayed in white on a blinking black background. The second is displayed in White on Dark Gray.

Declaration

```
procedure SetCrtBorder(Attr : Byte);
```

Purpose

Set screen border to Attr if card type and mode allow.

Description

SetCrtBorder allows you to set the color of the screen border to the specified video attribute (Attr) on color adapters (CGA, MCGA, EGA, VGA). It does nothing if the current display is a mono card. Be forewarned that the results are less than impressive on EGA cards.

Example

```
TextColor(White);
TextBackground(Blue);
ClrScr;
SetCrtBorder(Blue shl 4);
```

This sets the screen colors to white on blue, and the screen border to blue.

Declaration

```
procedure SetCursorSize(Startline, EndLine : Byte);
```

Purpose

Sets the cursor's starting and ending scan lines.

Description

Although other procedures in OPCRT allow you to change the size of the cursor, this is the only one that gives you complete control over the starting and ending scan lines. In monochrome text mode (7), the cursor box contains 14 scan lines, numbered from top (start) to bottom (end) as 0-13. In color text modes, the cursor box contains 8 scan lines, numbered from top to bottom as 0-7. Note that, although the EGA's cursor box has 14 scan lines, it normally emulates the CGA (for compatibility), and so can be treated as having only 8 scan lines in text modes.

Example

```
if CurrentMode = 7 then
  EndLine := $0D  {= 13}
else
  EndLine := $07; {= 7}
SetCursorSize(0, EndLine);
```

This would produce a "block cursor" (see the entry for BlockCursor).

See Also

BlockCursor
FatCursor
HiddenCursor
NormalCursor

SetCursorType / SetHercMode

Declaration

```
procedure SetCursorType(CT : CursorType);
```

Purpose

Sets the cursor shape to the specified type.

Description

SetCursorType combines the functions of four other routines--BlockCursor, HiddenCursor, FatCursor, and NormalCursor--into one. It is typically used in conjunction with ClassifyCursorType.

CT should be one of the following values: cuNormal, cuFat, cuBlock, or cuHidden. If CT is cuUnknown, the cursor will be hidden, just as it would if CT were cuHidden.

Example

```
var
  SaveCursorType : CursorType;
begin
  SaveCursorType := ClassifyCursorType;
  SetCursorType(cuHidden);
  ...
  SetCursorType(SaveCursorType);
end;
```

Save the current cursor type, hide the cursor, then restore the cursor to its previous shape.

See Also

BlockCursor
ClassifyCursorType
FatCursor
HiddenCursor
NormalCursor

Declaration

```
procedure SetHercMode(GraphMode : Boolean; GraphPage : Byte);
```

Purpose

Set Hercules card to graphics mode or text mode.

Description

This routine allows you to switch a Hercules card into graphics mode (GraphMode = True) or text mode (GraphMode = False). If GraphMode is True, GraphPage specifies which graphics page is to be made the active page (0 or 1). The GraphPage parameter is ignored if GraphMode is False.

Example

```
HercGraph := HercGraphicsMode;
if HercGraph then SetHercMode(False, 0);
...
if HercGraph then SetHercMode(True, 0);
```

Switch a Hercules graphics card into text mode, then back into graphics mode.

See Also

HercGraphicsMode
HercPresent

Declaration
```
procedure SetVisiblePage(PageNum : Byte);
```

Purpose
Set current video page.

Description
This routine allows you to set the current video page on a color adapter (CGA, EGA, MCGA, or VGA). It has no effect on monochrome adapters. In 40-column text modes, there are 8 video pages available (numbered 0-7). In 80-column text modes, there are 4 (numbered 0-3). Note that the current video page is always stored in the global variable CurrentPage.

Note that this function does *not* do the same thing as Graph.SetVisiblePage.

Example
```
if CurrentMode = 7 then
  WriteLn('Color adapter required')
else begin
  WriteLn('This is video page 0'); Delay(1000);
  SetVisiblePage(1);
  ClrScr;
  WriteLn('This is video page 1'); Delay(1000);
  SetVisiblePage(0);
end;
```

See Also
SetActivePage

Declaration
```
procedure StoreWindowCoordinates(var WC : WindowCoordinates);
```

Purpose
Store the window coordinates for the active window.

Description
This routine allows you to save the window coordinates for the active window. They can later be restored using RestoreWindowCoordinates.

Example
```
var WC: WindowCoordinates;
begin
  StoreWindowCoordinates(WC);
  Window(1, 1, 80, 10);
  ...
  RestoreWindowCoordinates(WC);
end;
```

See Also
RestoreWindowCoordinates

StuffKey / StuffString

Declaration

```
procedure StuffKey(W : Word);
```

Purpose

Stuff one key into the keyboard buffer.

Description

This routine inserts the specified key code directly into the keyboard buffer. If the keyboard buffer is full, it does nothing.

Examples

```
StuffKey($3B00);
```

Inserts the key code for <F1> into the keyboard buffer.

```
StuffKey(Ord('A'));
```

Inserts the key code for <A> into the keyboard buffer. (Most programs ignore the high byte of the key code if the low byte is non-zero.)

Declaration

```
procedure StuffString(S : string);
```

Purpose

Stuff the contents of S into the keyboard buffer.

Description

This routine inserts the specified string directly into the keyboard buffer. The string must be a maximum of 15 characters long (assuming that the keyboard buffer is empty to begin with). Excess characters are ignored. Along with StuffKey, this routine can be used to implement a simple keyboard macro system, as well as to pass keystrokes to programs being executed with Exec (DOS) or ExecDos (OPDOS).

Example

```
StuffString('Hello world!'^M);
ReadLn(MyString);
```

This would load MyString with 'Hello world!'.

Declaration

```
procedure SwitchInColorCard(ColorOn : Boolean);
```

Purpose

Activate or deactivate colors on a Hercules InColor card.

Description

This routine activates or deactivates CGA emulation on a Hercules InColor Card. It should be called only if the global variable WhichHerc is set to HercInColor.

Note that this routine is called automatically by the OPCRT initialization code in order to activate CGA emulation. Consequently, all you need to do to support the InColor card in most programs is to check the value returned by UseColor when selecting the colors you want to display text in. If it returns True, then you should select the same colors that you would use on a CGA or other color card, even if the current video mode is 7.

Getting back to our subject proper, SwitchInColorCard needs to be called only if you wish to deactivate CGA emulation, or if you need to reenable it after executing another program (for example).

Example

```
if WhichHerc = HercInColor then
  SwitchInColorCard(False);
{... execute the other program ...}
if WhichHerc = HercInColor then
  SwitchInColorCard(True);
```

Switch back to monochrome colors, execute another program, then reenable CGA emulation.

Declaration

```
function UseColor : Boolean;
```

Purpose

Returns True if color video attributes should be used.

Description

This function determines whether or not color video attributes should be selected when given a choice between color and monochrome (black and white). UseColor bases its decision on several factors. If the global typed constant DefColorChoice is set to ForceColor or ForceMono, UseColor returns True or False, respectively. If DefColorChoice is set to UseDefault, as it is by default, then the choice is made based primarily on the current video mode: if the mode is 0 (BW40) or 2 (BW80), it returns False; if the mode is 7 (Mono), it will return False unless the global variable WhichHerc is set to HercInColor; otherwise, it returns True.

Example

```
TextColor(White);
if UseColor then
  TextBackground(Blue)
else
  TextBackground(Black);
ClrScr;
```

Fill the screen with blue on color displays, black on monochrome displays.

See Also

ColorMono

WhereXAbs / WhereXY

Declaration

```
function WhereXAbs : Byte;
```

Purpose

Return absolute X coordinate (column) of cursor.

Description

Like the standard WhereX function, this routine returns the current X coordinate (column) of the cursor, but the number is absolute, rather than relative to the current window. As with WhereX, all X coordinates returned are 1-based (top left corner of screen is 1,1).

Example

```
Window(3,3,80,25);
GotoXY(1,1);
```

In this case, WhereX would return 1, and WhereXAbs would return 3.

See Also

WhereYAbs

Declaration

```
function WhereXY : Word;
```

Purpose

Returns the absolute X (column) and Y (row) coordinates of the cursor.

Description

This routine combines the functions of WhereXAbs and WhereYAbs, allowing you to get both values with a single call. The high byte of the result has the current Y coordinate (row), low byte has the current X coordinate (column). As with WhereXAbs and WhereYAbs, all coordinates returned are 1-based (top left corner of screen is 1,1).

Example

```
Window(3,3,80,25);
GotoXY(1,1);
```

In this case, WhereX and WhereY would return 1, WhereXAbs and WhereYAbs would return 3, and WhereXY would return $0303 (771).

See Also

WhereYAbs WhereXYdirect

Declaration

```
procedure WhereXYdirect(var X, Y : Byte);
```

Purpose

Read the current position of the cursor directly from the CRT controller.

Description

Unlike other routines in OPCRT that read the position of the cursor, this one reads the cursor position directly from the 6845 CRT controller using Port instructions, bypassing the BIOS. Obviously, this makes WhereXYdirect extremely hardware dependent, but it also allows you to read the cursor position accurately in cases where other routines fail. The most common application for WhereXYdirect is in TSRs, which are sometimes asked to pop up over applications that do not use the BIOS to position the cursor. The best known example of such a program is Lotus 1-2-3, but there are many others.

Example

```
var
  SaveX, SaveY : Byte;
begin
  {get the cursor position}
  WhereXYdirect(SaveX, SaveY);
  {update the BIOS's cursor data}
  GotoXYabs(SaveX, SaveY);
  ...
  {restore the cursor to its previous position}
  GotoXYabs(SaveX, SaveY);
end;
```

See Also

GotoXYabs WhereXY

Declaration

```
function WhereYAbs : Byte;
```

Purpose

Returns absolute Y coordinate (row) of the cursor.

Description

Like the standard WhereY function, this routine returns the current Y coordinate (row) of the cursor, but the number is absolute, rather than relative to the current window. As with WhereY, all Y coordinates returned are 1-based (top left corner of screen is 1,1).

Example

```
Window(3,3,80,25);
GotoXY(1,1);
```

In this case, WhereY would return 1, and WhereYAbs would return 3.

See Also

WhereXAbs WhereXY

WriteAttribute

Declaration

```
procedure WriteAttribute(St : String; Row, Col : Word);
```

Purpose

Write string of attributes St at Row,Col without changing characters.

Description

This routine takes as input a string of video attributes and writes them to the screen starting at Row,Col. The characters on the screen are not disturbed. WriteAttribute is intended to be used in conjunction with ReadAttribute.

Example

```
ReadAttribute(ScreenWidth, 1, 1, S);
for I := 1 to ScreenWidth do
  Byte(S[I]) := SwapNibble(Byte(S[I])) and $7F;
WriteAttribute(S, 1, 1);
```

Reverses all the foreground and background colors on row 1.

See Also

ReadAttribute

2. The ColorSet

The one object type defined by OPCRT is the ColorSet, and it is a very important one. The idea behind it is simple, but it is best understood by means of contrast.

In writing Turbo Professional, the predecessor of **Object Professional**, we frequently found ourselves struggling to find the best way to let programmers specify color choices when designing menus, pick lists, data entry screens, and other screen-oriented constructs. Over time, the number of methods for specifying choices proliferated: some routines took a series of video attributes as parameters; others took a single, special-purpose array of video attributes as a parameter; others required global variables to be modified; and so on.

When we started working on **Object Professional**, we decided that a major change in strategy was in order. Because most programs tend to use the same colors over and over for similar things (window frames, window headers, text within windows, etc.), we wanted to allow color choices to be centralized in a single data structure that could be reused throughout the program. We also wanted to simplify the task of writing programs that look good on both color and monochrome displays. The programmer, we thought, should be able to specify a pair of alternative video attributes, and have the library routine choose the appropriate one at runtime.

The solution we came up with is the ColorSet, an object that contains 31 pairs of video attributes, specifying color preferences for everything from window frames to delimiters for protected data entry fields to marked blocks in a text editor. Unlike other objects in **Object Professional**, a ColorSet doesn't really "do" anything. Although it has numerous methods (all static), the methods do nothing more than change the values of its data fields. The job of a ColorSet is simply to act as a receptacle, a place to store color choices, which can be conveniently passed to other objects as a group.

A ColorSet will almost always be declared in the form of a typed constant such as the following:

```
const MyColorSet : ColorSet = (
  TextColor       : $1E; TextMono       : $0F;
  CtrlColor       : $1C; CtrlMono       : $07;
  FrameColor      : $13; FrameMono      : $07;
  HeaderColor     : $3F; HeaderMono     : $70;
  ShadowColor     : $08; ShadowMono     : $70;
  HighlightColor  : $4F; HighlightMono  : $70;
  PromptColor     : $30; PromptMono     : $07;
  SelPromptColor  : $30; SelPromptMono  : $07;
  ProPromptColor  : $30; ProPromptMono  : $07;
  FieldColor      : $31; FieldMono      : $0F;
  SelFieldColor   : $31; SelFieldMono   : $0F;
  ProFieldColor   : $17; ProFieldMono   : $07;
  ScrollBarColor  : $13; ScrollBarMono  : $07;
  SliderColor     : $13; SliderMono     : $0F;
  HotSpotColor    : $30; HotSpotMono    : $70;
  BlockColor      : $3E; BlockMono      : $0F;
  MarkerColor     : $5F; MarkerMono     : $70;
  DelimColor      : $31; DelimMono      : $0F;
  SelDelimColor   : $31; SelDelimMono   : $0F;
  ProDelimColor   : $31; ProDelimMono   : $0F;
  SelItemColor    : $3E; SelItemMono    : $70;
  ProItemColor    : $17; ProItemMono    : $07;
  HighItemColor   : $1F; HighItemMono   : $0F;
  AltItemColor    : $1F; AltItemMono    : $0F;
  AltSelItemColor : $3F; AltSelItemMono : $70;
```

```
    FlexAHelpColor  : $1F; FlexAHelpMono  : $0F;
    FlexBHelpColor  : $1F; FlexBHelpMono  : $0F;
    FlexCHelpColor  : $1B; FlexCHelpMono  : $70;
    UnselXrefColor  : $1E; UnselXrefMono  : $09;
    SelXrefColor    : $5F; SelXrefMono    : $70;
    MouseColor      : $4F; MouseMono      : $70);
```

The values on the left (XxxColor) specify the video attributes to be used on color displays; those on the right (XxxMono) are to be used on monochrome or black and white displays.

As indicated above, the only thing you can really do with a ColorSet is pass it as a parameter to another object's constructor. For example:

```
MyStackWindow.InitCustom(2, 2, 79, 24, MyColorSet, wBordered)
```

In most cases, the constructor will make copies of all of the pairs of video attributes it needs to do its job. When one of the object's other methods needed to display something on the screen, it would then pass the two choices (XxxColor and XxxMono) to the ColorMono routine in OPCRT, and ColorMono would return the correct choice for the current display adapter.

The entry below for ColorSet gives a complete list of all its data fields, and a brief description of what each is generally used for. In many cases, the use for a given field is largely self-explanatory. FrameColor and FrameMono, for example, are the video attributes to be used when drawing a frame around a window.

In other cases, however, the use of a given field is not fully explained here. These cases fall into two categories. In the first category are values used by only one or two objects, such as FlexAHelpColor/FlexAHelpMono (used only in OPHELP). The full meaning of these values is explained in the documentation for the object(s) that use them. In the second category are values that are used in slightly differently ways by different objects. The most notable case is that of TextColor/TextMono. For any window-based object, these indicate the attribute to use when clearing the interior portion of the window; any text written within the window using Write statements will appear in that attribute. But some objects, such as PickLists and Menus, also use TextColor/TextMono for a second purpose. These secondary purposes are explained in the documentation for the object in question.

Types

```
ColorSet =
  object
    TextColor, TextMono : Byte;
      {Ordinary text; interior portion of a window; unselected pick/menu items}
    CtrlColor, CtrlMono : Byte;
      {Control characters, when displayed as uppercase alphas}
    FrameColor, FrameMono : Byte;
      {Window frames}
    HeaderColor, HeaderMono : Byte;
      {Window headers}
    ShadowColor, ShadowMono : Byte;
      {Window shadows}
    HighlightColor, HighlightMono : Byte;
      {Color used to highlight text found by a TextEditor}
    PromptColor, PromptMono : Byte;
      {Prompts for unselected data entry fields}
    SelPromptColor, SelPromptMono : Byte;
      {Prompts for selected data entry fields}
    ProPromptColor, ProPromptMono : Byte;
      {Prompts for protected data entry fields}
```

```
    FieldColor, FieldMono : Byte;
      {Unselected data entry fields}
    SelFieldColor, SelFieldMono : Byte;
      {Selected data entry fields}
    ProFieldColor, ProFieldMono : Byte;
      {Protected data entry fields}
    ScrollBarColor, ScrollBarMono : Byte;
      {Scroll bars}
    SliderColor, SliderMono : Byte;
      {Slider in scroll bars}
    HotSpotColor, HotSpotMono : Byte;
      {Mouse hot spots}
    BlockColor, BlockMono : Byte;
      {Marked blocks in a TextEditor, Browser}
    MarkerColor, MarkerMono : Byte;
      {Text markers in TextEditor, Browser}
    DelimColor, DelimMono : Byte;
      {Delimiters for unselected data entry fields}
    SelDelimColor, SelDelimMono : Byte;
      {Delimiters for selected data entry fields}
    ProDelimColor, ProDelimMono : Byte;
      {Delimiters for protected data entry fields}
    SelItemColor, SelItemMono : Byte;
      {Selected pick and menu items}
    ProItemColor, ProItemMono : Byte;
      {Protected pick and menu items}
    HighItemColor, HighItemMono : Byte;
      {Highlight characters in menu items}
    AltItemColor, AltItemMono : Byte;
      {Alternate color for unselected pick items}
    AltSelItemColor, AltSelItemMono : Byte;
      {Alternate color for selected pick items}
    FlexAHelpColor, FlexAHelpMono : Byte;
      {Highlighted help text surrounded by ^A}
    FlexBHelpColor, FlexBHelpMono : Byte;
      {Highlighted help text surrounded by ^B}
    FlexCHelpColor, FlexCHelpMono : Byte;
      {Highlighted help text surrounded by ^C}
    UnselXrefColor, UnselXrefMono : Byte;
      {Unselected cross reference in a help screen}
    SelXrefColor, SelXrefMono : Byte;
      {Selected cross reference in a help screen}
    MouseColor, MouseMono : Byte;
      {Color for mouse cursor; a placeholder, never used directly}
  end;
```

A collection of video attributes to be used by other objects.

SetXxxAttr

Declaration

```
procedure SetXxxAttr(Color, Mono : Byte);
```

Purpose

Set one pair of video attributes.

Description

All of the methods for a ColorSet are of this form, where 'Xxx' is the name of the attributes to be modified. A complete list of all the methods appears below. Color is the video attribute to be used for color displays; Mono is the attribute for monochrome or black and white displays. Note that, if Mono is $FF (= AutoMapMono), the monochrome attribute will be selected automatically based on the conversion table ColorMap (i.e., Mono = ColorMap[Color]). The actual methods are:

Method	Attribute used for
SetBlockAttr	Marked blocks
SetCtrlAttr	Control characters
SetDelimAttr	Unselected entry field delimiters
SetFieldAttr	Unselected entry fields
SetFlexAHelpAttr	Help text surrounded by ^A
SetFlexBHelpAttr	Help text surrounded by ^B
SetFlexCHelpAttr	Help text surrounded by ^C
SetFrameAttr	Window frames
SetHeaderAttr	Window headers
SetHighlightAttr	Highlight found text in browser/editor
SetHighlightItemAttr	Highlighted selection characters for menu items
SetHotSpotAttr	Mouse hot spots
SetMarkerAttr	Text markers
SetMouseAttr	Mouse cursor
SetPromptAttr	Entry field prompts
SetProtectedDelimAttr	Protected entry field delimiters
SetProtectedFieldAttr	Protected entry fields
SetProtectedItemAttr	Protected pick and menu items
SetProtectedPromptAttr	Protected entry field prompts
SetScrollBarAttr	Scroll bars
SetSelectedDelimAttr	Selected entry field delimiters
SetSelectedFieldAttr	Selected entry field fields
SetSelectedItemAttr	Selected pick and menu items
SetSelectedPromptAttr	Selected entry field prompts
SetSelXrefAttr	Selected cross references in help text
SetShadowAttr	Window shadows
SetSliderAttr	Sliders in scroll bars
SetTextAttr	Ordinary text
SetUnselXrefAttr	Unselected cross references in help text

Examples

```
MyColorSet.SetTextAttr($1F, $0F);
```

Set attributes for normal text to white on blue for color displays, white on black for monochrome displays.

```
MyColorSet.SetTextAttr($1F, AutoMapMono);
MyColorSet.SetTextAttr($1F, $FF);
```

These two statements are functionally identical. Both request that a monochrome attribute be selected automatically. In this particular example, the monochrome attribute would be set to $0F (= ColorMap[$1F]).

C. OPCOLOR: Color Names

This unit is provided strictly as a convenience for users who would prefer to use named constants, rather than hexadecimal numbers, to refer to video attributes. Use of this unit is entirely optional. A complete list of the constants defined in OPCOLOR appears below.

Within **Object Professional**, OPCOLOR is used only by the MAKEMENU and MAKESCRN utilities, both of which can optionally generate code that relies on the color constants in OPCOLOR.

Note that OPCOLOR does not define any constants for video attributes greater than $7F. You can, however, achieve the desired effect by adding '+Blink' to the end of the color name. For example, to create blinking white-on-red text, you would use the expression 'WhiteOnRed+Blink'. (Blink is a constant defined in the OPCRT unit.)

Constants

```
{Black background}
BlackOnBlack          = $00;
BlueOnBlack           = $01;
GreenOnBlack          = $02;
CyanOnBlack           = $03;
RedOnBlack            = $04;
MagentaOnBlack        = $05;
BrownOnBlack          = $06;
LtGrayOnBlack         = $07;
DkGrayOnBlack         = $08;
LtBlueOnBlack         = $09;
LtGreenOnBlack        = $0A;
LtCyanOnBlack         = $0B;
LtRedOnBlack          = $0C;
LtMagentaOnBlack      = $0D;
YellowOnBlack         = $0E;
WhiteOnBlack          = $0F;
{Blue background}
BlackOnBlue           = $10;
BlueOnBlue            = $11;
GreenOnBlue           = $12;
CyanOnBlue            = $13;
RedOnBlue             = $14;
MagentaOnBlue         = $15;
BrownOnBlue           = $16;
LtGrayOnBlue          = $17;
DkGrayOnBlue          = $18;
LtBlueOnBlue          = $19;
LtGreenOnBlue         = $1A;
LtCyanOnBlue          = $1B;
LtRedOnBlue           = $1C;
LtMagentaOnBlue       = $1D;
YellowOnBlue          = $1E;
WhiteOnBlue           = $1F;
{Green background}
BlackOnGreen          = $20;
BlueOnGreen           = $21;
GreenOnGreen          = $22;
CyanOnGreen           = $23;
RedOnGreen            = $24;
MagentaOnGreen        = $25;
```

```
BrownOnGreen          = $26;
LtGrayOnGreen         = $27;
DkGrayOnGreen         = $28;
LtBlueOnGreen         = $29;
LtGreenOnGreen        = $2A;
LtCyanOnGreen         = $2B;
LtRedOnGreen          = $2C;
LtMagentaOnGreen      = $2D;
YellowOnGreen         = $2E;
WhiteOnGreen          = $2F;
{Cyan background}
BlackOnCyan           = $30;
BlueOnCyan            = $31;
GreenOnCyan           = $32;
CyanOnCyan            = $33;
RedOnCyan             = $34;
MagentaOnCyan         = $35;
BrownOnCyan           = $36;
LtGrayOnCyan          = $37;
DkGrayOnCyan          = $38;
LtBlueOnCyan          = $39;
LtGreenOnCyan         = $3A;
LtCyanOnCyan          = $3B;
LtRedOnCyan           = $3C;
LtMagentaOnCyan       = $3D;
YellowOnCyan          = $3E;
WhiteOnCyan           = $3F;
{Red background}
BlackOnRed            = $40;
BlueOnRed             = $41;
GreenOnRed            = $42;
CyanOnRed             = $43;
RedOnRed              = $44;
MagentaOnRed          = $45;
BrownOnRed            = $46;
LtGrayOnRed           = $47;
DkGrayOnRed           = $48;
LtBlueOnRed           = $49;
LtGreenOnRed          = $4A;
LtCyanOnRed           = $4B;
LtRedOnRed            = $4C;
LtMagentaOnRed        = $4D;
YellowOnRed           = $4E;
WhiteOnRed            = $4F;
{Magenta background}
BlackOnMagenta        = $50;
BlueOnMagenta         = $51;
GreenOnMagenta        = $52;
CyanOnMagenta         = $53;
RedOnMagenta          = $54;
MagentaOnMagenta      = $55;
BrownOnMagenta        = $56;
LtGrayOnMagenta       = $57;
DkGrayOnMagenta       = $58;
LtBlueOnMagenta       = $59;
LtGreenOnMagenta      = $5A;
LtCyanOnMagenta       = $5B;
LtRedOnMagenta        = $5C;
LtMagentaOnMagenta    = $5D;
```

```
YellowOnMagenta      = $5E;
WhiteOnMagenta       = $5F;
{Brown background}
BlackOnBrown         = $60;
BlueOnBrown          = $61;
GreenOnBrown         = $62;
CyanOnBrown          = $63;
RedOnBrown           = $64;
MagentaOnBrown       = $65;
BrownOnBrown         = $66;
LtGrayOnBrown        = $67;
DkGrayOnBrown        = $68;
LtBlueOnBrown        = $69;
LtGreenOnBrown       = $6A;
LtCyanOnBrown        = $6B;
LtRedOnBrown         = $6C;
LtMagentaOnBrown     = $6D;
YellowOnBrown        = $6E;
WhiteOnBrown         = $6F;
{Light gray background}
BlackOnLtGray        = $70;
BlueOnLtGray         = $71;
GreenOnLtGray        = $72;
CyanOnLtGray         = $73;
RedOnLtGray          = $74;
MagentaOnLtGray      = $75;
BrownOnLtGray        = $76;
LtGrayOnLtGray       = $77;
DkGrayOnLtGray       = $78;
LtBlueOnLtGray       = $79;
LtGreenOnLtGray      = $7A;
LtCyanOnLtGray       = $7B;
LtRedOnLtGray        = $7C;
LtMagentaOnLtGray    = $7D;
YellowOnLtGray       = $7E;
WhiteOnLtGray        = $7F;
```

D. OPMOUSE: Mouse Routines

In writing OPMOUSE, we aspired to three main goals. First, we wanted it to provide routines that were optimized for mouse programming in text mode, in part because other collections of mouse routines that we had seen tend to be weak in this area, but also because all the other screen-oriented routines in **Object Professional** are designed to be used in text mode. Second, we wanted these routines to provide a high-level interface, insulating the programmer as much as possible from the mouse driver. And third, we wanted it to provide a body of routines that we could build on in other **Object Professional** units, insulating the programmer still further.

Getting Started with OPMOUSE

As you begin working with OPMOUSE, you should be pleased to learn that we have chosen to model it after OPCRT, a unit that you're probably already familiar with. Besides reducing the learning curve for OPMOUSE, this has some other benefits that we'll be pointing out later. To give you an idea of the extent to which the model has been followed, here's a quick list of rough correspondences:

OPMOUSE routine	OPCRT "equivalent"
MousePressed	KeyPressed
MouseKeyWord	ReadKeyWord
InitializeMouse	ReinitCrt
MouseWhereX	WhereX
MouseWhereXabs	WhereXabs
MouseWhereY	WhereY
MouseWhereYabs	WhereYabs
MouseWhereXY	WhereXY
MouseGotoXY	GotoXY
MouseWindow	Window
RestoreMouseCoordinates	RestoreWindowCoordinates
StoreMouseCoordinates	StoreWindowCoordinates
NormalMouseCursor	NormalCursor
FatMouseCursor	FatCursor
BlockMouseCursor	BlockCursor
HiddenMouseCursor	HiddenCursor
GetMousePage	CurrentPage (global variable)
SetMousePage	SetVisiblePage

With a few significant exceptions, most of the other routines in OPMOUSE are ones that you won't need very often. If you're already up to speed with OPCRT, you should therefore have little difficulty in getting familiar with the basic tools provided in OPMOUSE.

One important point to note here is that you should, in general, make sure that a mouse is installed before calling any of the routines in OPMOUSE. The call will simply do nothing if a mouse is not installed. But if the routine in question returns results in VAR parameters, for example, those values will be uninitialized upon return.

Mouse Support in Other Units

As mentioned earlier, many of the objects in **Object Professional** provide integral mouse support based on the routines in OPMOUSE: menus, pick lists, entry screens, text editors–any object that uses a CommandProcessor supports the mouse. In each case, you can enable this support simply by activating an option in the appropriate command processor, like this

```
XxxCommands.cpOptionsOn(cpEnableMouse);
```

where XxxCommands is the name of the CommandProcessor associated with the object.

In most cases, all you need to do before enabling mouse support is to select the kind of mouse cursor that you want to use (see the discussion of "The Mouse Cursor," below). You may also want to make the mouse cursor visible at this point, by calling ShowMouse, but that isn't always necessary. Once mouse support is enabled in these units, the routines in them will make sure that the mouse cursor is on when they need it to be, and restore it to its previous state on exit.

Although you will usually want to provide additional forms of mouse support in your own code, it is actually possible to write a program that makes effective use of the mouse simply by adding a few procedure/method calls to the beginning of the program. If you examine the demo programs EDITOR.PAS and ENTRY.PAS, paying special attention to sections of code surrounded by '{$IFDEF UseMouse}' and '{$ENDIF}', you'll see that remarkably little effort was required in order to add mouse support to these programs.

The Mouse Cursor

In the previous paragraph, we indicated that you need to choose the kind of mouse cursor that you want to use. If you weren't aware of it before, mouse cursors come in two varieties, hard and soft. When you choose a hardware mouse cursor, the mouse driver simply moves the "real" cursor around the screen in response to the movements of the mouse. (Hereafter, we'll refer to the "real" cursor as "the system cursor".) When you choose a software cursor, the mouse driver leaves the system cursor alone and instead manipulates characters on the screen (temporarily changes their color and/or overwrites them with another character) in order to indicate the position of the mouse cursor. When the mouse is moved, the character beneath the mouse cursor is restored, and another location on the screen is altered.

If you're having a hard time picturing the difference, try running the following program:

```
program MouseCursors;
uses OpCrt, OpMouse;
begin
  if not MouseInstalled then Halt;
  NormalCursor;
  ClrScr;
  FastCenter('Click the left button to change cursors...', 25, $07);
  EnableEventHandling;
  SoftMouseCursor($0000, (ColorMono($4F, $70) shl 8)+$04);
  ShowMouse;
  while MouseKeyWord <> MouseLft do {nothing} ;
  HideMouse;
  FastCenter('Click the left button to exit...', 25, $0F);
  BlockMouseCursor;
  ShowMouse;
  while MouseKeyWord <> MouseLft do {nothing} ;
  NormalCursor;
end.
```

When the program starts, the mouse cursor will be represented as a diamond (red and white on color systems, reverse video on mono systems), and the system cursor will sit blinking at the top left corner of the screen. This is an example of a "soft" mouse cursor, a cursor generated through software (the mouse driver). After you press the left mouse button once, the diamond will disappear, the system cursor will change to a blinking block, and any movement of the mouse will cause the system cursor to be moved. This is an example of a "hard" mouse cursor, a cursor generated through hardware (the display adapter).

A hardware mouse cursor is the best choice for most programs in which it is an option. Unfortunately, those programs are rare. Most programs that have a use for the mouse also

have a use for the system cursor, and can't afford to have it moved each time the user moves the mouse. But if you do have a program that fits the bill, the only real decision you have to make is the shape of the cursor. You can use the HardMouseCursor routine, which lets you specify the starting and ending scan lines for the cursor, or you can use one of the higher level routines: NormalMouseCursor, FatMouseCursor, or BlockMouseCursor.

A word of warning about the hardware mouse cursor. The mouse driver positions it by programming the CRT controller, and it does not update the BIOS data area, so you cannot determine its position using WhereX, WhereY, etc. The only routine in OPCRT that will return the position of a hardware mouse cursor is WhereXYdirect. For related reasons, the cursor positioning routines in OPCRT (GotoXY, GotoXYabs) should not be used to move a hardware mouse cursor.

A software cursor will be your choice not only in cases where use of a hardware cursor isn't feasible, but also in cases where you want to achieve a special effect that isn't possible with a hardware cursor. The default mouse cursor is a sort of non-blinking/see-through/reverse-video block, whose color changes as it moves in order to assure that its color always contrasts with that of the character beneath it. How is this possible?

When you select a software cursor (using SoftMouseCursor), you must specify a "screen mask" and a "cursor mask". The easiest way to understand how these values are used is to study a couple of examples. Let's say that the mouse cursor is about to be moved on top of a character, 'A' (ASCII #$41), which is being displayed in white on black (a video attribute of $0F). In video memory, this character is represented as a word, with the attribute in the high byte: $0F41. Given the default screen mask of $FFFF (= DefaultScreenMask) and the default cursor mask of $7700 (= DefaultCursorMask), the mouse driver would then create the mouse cursor by first ANDing the character value with the screen mask

```
$0F41 AND $FFFF  =  $0F41 <-- no change
```

and then XORing the result with the cursor mask:

```
$0F41 XOR $7700  =  $7841 <-- attribute changed, character unchanged
```

This would produce an 'A' in reverse video (dark gray on light gray). Now let's say you wanted a slightly different effect, variable colors with a mouse cursor shaped like a diamond (ASCII #$04). In this case you'd use a screen mask of $FF00 and a cursor mask of $7704:

```
$0F41 AND $FF00  =  $0F00 <-- attribute unchanged, character wiped out
$0F00 XOR $7704  =  $7804 <-- attribute changed, new character in low byte
```

Finally, let's say that you want the mouse cursor to look the same no matter where it is on the screen: you always want want it to appear as a musical note (ASCII #$14) displayed in white on red ($4F). In this case you'd use a screen mask of $0000 and a cursor mask of $4F14:

```
$0F41 AND $0000  =  $0000 <-- attribute and character both wiped out
$0000 XOR $4F14  =  $4F14 <-- new attribute, new character
```

Reading The Mouse Buttons

When you start adding mouse support to your programs, you'll soon discover that it really isn't that difficult if you have the right data at your disposal. In most cases, that means that you need to know one or both of two things: where the mouse is, and what mouse button(s) if any have been pressed. In this section, we'll concentrate on the mouse buttons.

As we indicated, OPMOUSE is modelled on the OPCRT unit. So it should be no surprise that two of the most basic functions in OPMOUSE are MousePressed and MouseKeyWord, which

are to the mouse buttons what KeyPressed and ReadKeyWord are to keys. MousePressed tells you if mouse button(s) are being pressed, and MouseKeyWord returns a value indicating what was pressed. Before using them, there are only two things you need to know. First, these functions can do nothing until EnableEventHandling has been called. We'll explain why in "Mouse Event Handlers." Second, MouseKeyWord returns what we call a "pseudo-scan code," a word value that looks just like a scan code returned by ReadKeyWord:

```
MouseLft    = $EF00; {left button}
MouseRt     = $EE00; {right button}
MouseBoth   = $ED00; {both buttons}
MouseCtr    = $EC00; {center button}
MouseLftCtr = $EB00; {left and center buttons}
MouseRtCtr  = $EA00; {right and center buttons}
MouseThree  = $E900; {all three buttons}
```

As you can see, the low byte is always 0, and the high byte indicates which button or buttons have been pressed. Among other things, this system makes it easy to add mouse support to programs that are already set up to handle keyboard input. (Note that these pseudo-scan codes are displayed by DESKPOP's quick reference chart.)

That raises the question, "Wouldn't it be easier if a single function returned either a key or a mouse button?" Yes indeed, and the function is ReadKeyOrButton. This routine simply sits in a loop waiting for a key or mouse button to be pressed, then returns either the scan code for the key, if a key was pressed first, or the pseudo-scan code for the mouse button, if a button was pressed first. (It also returns, via the global variables MouseKeyWordX and MouseKeyWordY, the location of the mouse cursor at the time the key was pressed.) If you have a program that currently uses ReadKeyWord to obtain keyboard input, you can thus add some degree of mouse support just by replacing the call to ReadKeyWord with one to ReadKeyOrButton, and adding some code to handle the button presses. (You would also probably want to replace all calls to KeyPressed with calls to KeyOrButtonPressed.)

There are other routines in OPMOUSE that return information about the mouse buttons, but in our experience these are the most useful. Why? Because the other routines can only tell you what buttons are being pressed at any given moment. They don't guarantee that those buttons have also been released. MouseKeyWord and ReadKeyOrButton do, provided that the global variable WaitForButtonRelease is set to True. To see the importance of waiting for the button release, try running some experiments using ReadKeyOrButton with WaitForButtonRelease set to False. What you'll find is that, in cases where more than one button is being pressed at a time, there are brief moments where only one of the buttons appears to be pressed, even though the user intended to press both buttons simultaneously. Only rarely will a program need WaitForButtonRelease to be set to False.

Mouse Windows and Mouse Coordinates

Also in keeping with the OPCRT model, OPMOUSE allows you to set up "mouse windows" (regions of the screen whose boundaries the mouse cursor must observe), and to read the current window-relative position of the mouse cursor.

To set up a mouse window, you call either of two routines: MouseWindow or FullMouseWindow. MouseWindow is just like Window. It takes four 1-based values as parameters–XLow, YLow, XHigh, YHigh–so to create an 80x25 window, you would use 'MouseWindow(1, 1, 80, 25);'. FullMouseWindow is a "shell routine": all it does is call MouseWindow with the current dimensions of the screen as parameters. Since OPMOUSE calls this procedure for you when your program starts, you usually don't need to call either

routine yourself unless you want to create a smaller window, or you have changed video modes, or switched in or out of the 8x8 font.

As mentioned in "Reading the Mouse Buttons," you can generally determine the location of the mouse cursor at the time a button was pressed by checking MouseKeyWordX and MouseKeyWordY. If you need to know its location at some other time, you have several options. One option is to use MouseWhereX and MouseWhereY, which are just like WhereX and WhereY. A second, and sometimes better option, is to use MouseWhereXY, which returns both the X and Y coordinates of the cursor and the status of the mouse buttons.

In each case, it is assumed that you want to know the position of the mouse cursor relative to the current mouse window. If that's not the case, you can use MouseWhereXabs and MouseWhereYabs to obtain the absolute coordinates for the mouse cursor at the moment. If you need to know where the mouse was when a button was pressed, you can easily calculate the absolute position by adding MouseXLo to the X (or column) coordinate, stored in MouseKeyWordX, and MouseYLo to the Y (or row) coordinate, stored in MouseKeyWordY. Along with MouseXHi and MouseYHi, MouseXLo and MouseYLo hold the current dimensions of the mouse window. (MouseXLo and MouseYLo are 0-based numbers, meaning that the top left corner of the screen is 0,0; MouseXHi and MouseYHi are 1-based.)

Mouse Droppings

Now for some bad news. Mouse programming can be extremely frustrating at times, because mice are not tidy animals, at least not when you are using a software cursor. Here's a typical scenario for you to consider:

The mouse cursor is being moved to row 1, column 1. Just before moving it, the mouse driver looks at the character-attribute pair at 1,1 and saves a copy of it. When the mouse gets moved again, the mouse driver will restore the pair before moving the cursor to its destination. As long as the contents of the screen don't change, this works fine, but what if your program writes something at row 1, column 1 while the mouse cursor is there? The mouse cursor would get overwritten, of course. That's not so bad, however, because the mouse cursor will be revealed again the next time it is moved. What is bad is that the mouse driver will then restore the character and attribute that used to be at 1,1. The result is an out-of-place character or oddball color on the screen, and if you aren't careful the screen can get littered with them. Because the effect can resemble the bottom of a mouse cage, these are aptly called "mouse droppings."

So, how do you prevent them from cluttering your carefully designed screens? Unfortunately, the only sure-fire method is to hide the mouse cursor *every* time you do a screen write. If the screen is about to be cleared, scrolled, painted, written to, saved or restored, you must call HideMouse or HideMousePrim beforehand and ShowMouse or ShowMousePrim afterward. This is why we said earlier that you will want to use a hardware cursor whenever possible.

Although there's nothing OPMOUSE can do to help you in your battles against mouse droppings, we can at least give you a tip that may prove useful in some situations. If you are doing a lot of screen I/O using a single routine, such as FastWrite, you might want to use a shell routine like this one:

```
procedure MouseWrite(S : String; Row, Col, Attr : Byte);
var SaveMouseCursor : Boolean;
begin
  HideMousePrim(SaveMouseCursor);
  FastWrite(S, Row, Col, Attr);
  ShowMousePrim(SaveMouseCursor);
end;
```

By using a routine like this, you can greatly reduce the number of changes that you have to make to your program. One global search-and-replace operation in your editor can take care of all the FastWrite operations in an entire unit. The price to be paid here is speed: screen writes take longer, and much time can be wasted turning the mouse cursor on and off, especially if you have a lot of screen writing to do. In cases where you have, for example, six successive calls to FastWrite, you will therefore probably want to just turn the mouse cursor off before the first one, and turn it on again after the last one.

Mouse Event Handlers

Although the high-level facilities that we've been discussing should be adequate for most applications, there may be times when you need information about what the mouse is doing as soon as an event takes place. (An "event" occurs when one of three things happen: the mouse is moved, a mouse button is pressed, or a mouse button is released.) To get this information, you need to write what is called a "mouse event handler."

It sounds complicated, but OPMOUSE makes the task quite easy. All you have to do is write a FAR-declared global procedure that takes no parameters, and call SetMouseEventHandler to tell OPMOUSE that you want it to be called when a mouse event occurs:

```
{$F+}
procedure MyEventHandler;
begin
  {handle event}
end;
...
SetMouseEventHandler(AllMouseEvents, @MyEventHandler);
```

OPMOUSE will then install its own low-level event handler, which takes the raw data provided by the mouse, massages it into usable form, stores it in global variables, then calls your routine.

SetMouseEventHandler takes two parameters. The second, UserRoutine, is the address of your event handler. The first, EventMask, is a bit-mapped byte that describes the events that you are interested in. You can construct this byte value using one or more of the following constants:

```
DisableEventHandler  = $00;
MouseMoved           = $01;
LeftButtonPressed    = $02;
LeftButtonReleased   = $04;
RightButtonPressed   = $08;
RightButtonReleased  = $10;
CenterButtonPressed  = $20; {three-button mice only}
CenterButtonReleased = $40; {three-button mice only}
AllMouseEvents       = $7F;
```

For example, if you were interested only in the left mouse button, you would specify 'LeftButtonPressed + LeftButtonReleased' as your EventMask. If you were interested in all events, you would use 'AllMouseEvents'. And if you wanted to disable your event handler at some point, you would use 'DisableEventHandler' (or pass 'nil' as the UserRoutine argument).

Once your event handler is installed, you can obtain all the information you need about the condition of the mouse by checking the following global typed constants:

```
MouseEvent  : MouseEventType = DisableEventHandler;
MouseStatus : ButtonStatus = NoButton;
MouseLastX  : Byte = 1;
MouseLastY  : Byte = 1;
```

MouseEvent is a bit-mapped byte of the same type as EventMask. You can check for a particular event, a mouse movement for example, by ANDing MouseEvent with the appropriate constant and checking for a non-zero result:

```
MouseWasMoved := MouseEvent and MouseMoved <> 0;
```

MouseStatus tells you which mouse buttons, if any, are being pressed. It is declared to be of the enumerated type ButtonStatus:

```
type
  ButtonStatus = (
    NoButton, LeftButton, RightButton, BothButtons,
    {the following values are possible only on a 3-button mouse}
    CenterButton, LeftAndCenterButtons, RightAndCenterButtons, All3Buttons);
```

And you can determine the window-relative location of the mouse cursor by looking at MouseLastX and MouseLastY.

When writing your event handler, keep in mind that events can take place many times a second. It needs to be short and sweet; it can't use much stack space; and it certainly can't call any procedures or functions that are not reentrant: that includes most of DOS, portions of the Turbo Pascal runtime library, and any routines of your own (or ours) that could be thrown off if the value of a global variable were to change unexpectedly in the middle of the routine.

Using the Mouse in a TSR

Writing a TSR that includes mouse support is not too difficult as long as you keep in mind a few key points. We'll spell these out in a moment, but we should probably warn you that this discussion assumes a working knowledge of how to write a TSR using the OPTSR (or OPSWAP) unit. If you don't have it, you should be sure to at least study the documentation for OPTSR before reading on. Better still, come back to this section after you've written at least one TSR that you're ready to add mouse support to. If you're already comfortable with TSRs, though, by all means read on.

The first point to bear in mind is that your TSR must be prepared to pop up over a program that is already using the mouse. To avoid disturbing it, you must be prepared to save the state of the mouse driver when you pop up, and restore it on exit. The routines you need to do this are MouseStateBufferSize, SaveMouseState, and RestoreMouseState.

MouseStateBufferSize is a function that returns the size of the buffer needed to store all the mouse driver's working data. We strongly recommend that you call it near the start of your program and allocate the buffer before going resident, for reasons that should be obvious to you in time. SaveMouseState and RestoreMouseState do exactly what their names suggest.

The second thing to keep in mind is that you cannot set up the mouse the way you want it before going resident and expect it to be configured that way when you pop up. The program you pop up over might have chosen a different kind of cursor or set up a tiny mouse window, for example. So you have to reinitialize the mouse each time you pop up.

Finally, if you need to use either ReadKeyOrButton, MouseKeyPressed, or MouseKeyWord, or if you have enabled mouse support for any of your objects, you will need to call EnableEventHandling on entry to your popup.

Now let's put some of these pieces together. The following is a sketchy outline of a TSR that uses OPMOUSE:

```
program MyTSR;
uses
  Dos, OpCrt, OpMouse, OpInt, OpTsr;
var
  MSP : MouseStatePtr;
  MSPsize : Word;

  {$F+}
  procedure PopupEntryPoint(var Regs : Registers);
  begin
    ReinitCrt;                        {reinitialize OPCRT}
    {** save screen and cursor **}
    if MouseInstalled then begin
      SaveMouseState(MSP, False);     {save the state of the mouse driver}
      InitializeMouse;                {reinitialize the mouse}
      EnableEventHandling;            {install event handler}
      {** set up a mouse window **}
      {** choose a mouse cursor and make it visible **}
    end;
    {** body of the popup goes here **}
    {** restore screen and cursor **}
    if MouseInstalled then
      RestoreMouseState(MSP, False); {restore state of the mouse driver}
  end;
  {$F-}

begin
  {...}
  if MouseInstalled then begin
    MSPsize := MouseStateBufferSize; {allocate mouse state buffer}
    {check for a reasonable buffer size}
    if (MSPsize = 0) or (MSPsize > 1000) then
      MouseInstalled := False
    else
      GetMem(MSP, MSPsize);           {allocate the buffer}
  end;
  {...}
end.
```

The only additional points worth making here are that you should be sure to 1) call ReinitCrt before saving the screen, 2) save the state of the mouse driver after saving the screen, and 3) restore the state of the mouse after restoring the screen. If you don't take these steps in the right order, you can easily corrupt the screen of the program you pop up over. If you need a more complete model than MyTSR, study the demonstration program DESKPOP.

Inequality among Mouse Drivers

Despite what was just said in the last paragraph, there is one more very important thing you need to know about MouseStateBufferSize, SaveMouseState, and RestoreMouseState. The function calls that they use aren't implemented by all mouse drivers. MicroSoft didn't have any need for them until they wrote CodeView, which had to be capable of debugging programs that used the mouse, so in the early versions of the MicroSoft mouse driver these functions didn't exist. They are probably implemented in the current versions of all mouse vendors' device drivers, but of course not everyone has the current versions.

The hard truth is that you have to be careful about using these functions (GetMousePage and SetMousePage fall into the same category) in your programs. We have attempted to make MouseStateBufferSize return 0 when it encounters a mouse driver that doesn't implement

them, but don't count on it. The value it returns is normally in the range 350-400, so it wise to check for unusually high values (greater than 1000, perhaps) as well as for 0. In addition, you should either provide your users with a means of disabling mouse support manually, or else issue stern warnings advising them that your program isn't guaranteed to work correctly with older mouse drivers.

A somewhat similar problem occurs (more rarely) with MouseGotoXY (see the entry for it, below). The moral of the story is that mouse drivers more than a year old are compatibility problems waiting to happen and should be replaced.

The Mickey

One last bit of technical data. The smallest unit of distance recognized by a mouse is a "mickey," roughly 1/100 of an inch. As you move a mouse across your desk, the mouse hardware must pass back to the mouse driver information that indicates how far the mouse has been moved, and this data is expressed in terms of mickeys. The mouse driver, in turn, must then determine if and how far the mouse cursor needs to be moved on the screen to reflect the changes in the position of the mouse. In making these determinations, it relies on a value known as the "mickey-to-pixel ratio."

This ratio indicates the number of mickeys that the mouse must be moved in order to move the mouse cursor eight pixels. Actually, however, there are two such ratios: one pertaining to horizontal movement, and one to vertical movement. The default ratios are 8 and 16, respectively. In 80-column text modes, the default settings thus require the mouse to be moved about 8/100 of an inch horizontally in order to move the mouse cursor one column to the left or right, and 16/100 of an inch up or down in order to move it by one row.

Confused? Fortunately, you don't need to understand much about the mechanics of mice in order to support them in your software. This section is provided primarily as background material for the discussions of the GetMickeyCount and SetMickeyToPixelRatio routines, described below, which most people will never need to use. (See the source code for the MAKEMISC unit, used by MAKEMENU and MAKESCRN, for an example of a case where SetMickeyToPixelRatio is useful.)

Types

```
ButtonStatus = (
  NoButton, LeftButton, RightButton, BothButtons,
  CenterButton, LeftAndCenterButtons, RightAndCenterButtons, All3Buttons);
```

Used in routines that check to see whether a particular mouse button, or combination of buttons, has been pressed. On a two-button mouse, only the values NoButton..BothButtons are meaningful.

```
MouseEventType = DisableEventHandler..AllMouseEvents;
```

A mouse event (mouse was moved, button was pressed, button was released).

```
MouseState =
  record
    BufSize : Word;
    Buffer  : array[1..400] of Byte;
  end;
MouseStatePtr = ^MouseState;
```

Data structure used to store the state of the mouse driver. Buffer is a variably-sized array, and may be larger than 400 bytes if necessary. BufSize is equal to the size of the array plus 2.

```
MouseXLo : Byte;
MouseXHi : Byte;
MouseYLo : Byte;
MouseYHi : Byte;
```

Screen coordinates for the current mouse window. As with the WindMin and WindMax variables in CRT and OPCRT, MouseXLo and MouseYLo are 0-based numbers, while MouseXHi and MouseYHi are 1-based numbers.

Constants

```
DefaultScreenMask = $FFFF;
DefaultCursorMask = $7700;
```

The default screen and cursor masks for a software mouse cursor.

```
DisableEventHandler  = $C0;
MouseMoved           = $01;
LeftButtonPressed    = $02;
LeftButtonReleased   = $04;
RightButtonPressed   = $08;
RightButtonReleased  = $10;
CenterButtonPressed  = $20;
CenterButtonReleased = $40;
AllMouseEvents       = $7F;
```

Bit masks for the various mouse events.

```
MouseEvent  : MouseEventType = DisableEventHandler;
MouseStatus : ButtonStatus = NoButton;
MouseLastX  : Byte = 1;
MouseLastY  : Byte = 1;
```

Used to pass information to event handlers that have been set up using SetMouseEventHandler.

```
MouseLft    = $EF00; {left button}
MouseRt     = $EE00; {right button}
MouseBoth   = $ED00; {both buttons}
MouseCtr    = $EC00; {center button}
MouseLftCtr = $EB00; {left and center buttons}
MouseRtCtr  = $EA00; {right and center buttons}
MouseThree  = $E900; {all three buttons}
```

These are the pseudo-scan codes returned by MouseKeyWord and ReadKeyOrButton when a mouse button (or combination of buttons) has been pressed.

```
MouseRoutine : Pointer = nil;
```

If not nil, MouseRoutine points to a user-written event handler set up using SetMouseEventHandler.

```
MouseRoutineEvent : MouseEventType = DisableEventHandler;
```

Bit-mapped byte indicating the event(s) that will cause the procedure pointed to by MouseRoutine to be called.

```
WaitForButtonRelease : Boolean = True;
```

If True, MouseKeyWord waits for the mouse button currently being pressed to be released before returning a pseudo-scan code.

Variables

```
MouseButtons : Byte;
```

Number of buttons on user's mouse (0, 2, or 3). Set by InitializeMouse.

```
MouseCursorOn : Boolean;
```

This variable keeps track of whether the mouse cursor is currently being displayed or not.

```
MouseInstalled : Boolean;
```

Indicates whether a mouse is installed or not.

```
MouseKeyWordX : Byte;
MouseKeyWordY : Byte;
```

These hold the coordinates of the mouse cursor at the time of the last call to MouseKeyWord.

Declaration

```
procedure BlockMouseCursor;
```

Purpose

Set scan lines for a block mouse cursor.

Description

This routine selects a hardware mouse cursor shaped like a large block. It is analogous to the BlockCursor routine in OPCRT.

See Also

BlockCursor (OPCRT) HardMouseCursor

Declaration

```
procedure DisableEventHandling;
```

Purpose

Disable the event handler installed by EnableEventHandling.

Description

This routine disables OPMOUSE's internal event handler, which is installed when EnableEventHandling is called. Its principal use is in TSRs that need to use MousePressed and MouseKeyWord. See the entry for EnableEventHandling for an example.

See Also

EnableEventHandling SetMouseEventHandler

EnableEventHandling / FatMouseCursor

Declaration
```
procedure EnableEventHandling;
```

Purpose
Enable the event handler needed for MousePressed and MouseKeyWord.

Description
This routine must be called before MousePressed or MouseKeyWord may be used in a program. What it does is to activate OPMOUSE's mouse event handler, a routine that will then be called by the mouse driver each time the mouse cursor is moved or a mouse button is pressed or released.

For more details, see the discussions on "Reading The Mouse Buttons" and "Mouse Event Handlers" at the beginning of the OPMOUSE section.

Example
```
EnableEventHandling;
...
if MousePressed then begin
  PseudoKey := MouseKeyWord;
  {handle command assigned to PseudoKey}
end;
```
Activate the event handler, then use MousePressed and MouseKeyWord to obtain mouse button information.

See Also
DisableEventHandling
MouseKeyWord
MousePressed
SetMouseEventHandler

Declaration
```
procedure FatMouseCursor;
```

Purpose
Set larger scan lines for mouse cursor based on current video mode.

Description
This routine selects a hardware mouse cursor that is slightly larger than a normal two-line cursor. It is analogous to the FatCursor routine in OPCRT.

See Also
FatCursor (OPCRT)
HardMouseCursor

Declaration

```
procedure FullMouseWindow;
```

Purpose

Sets mouse window coordinates to full screen.

Description

This routine selects the entire screen as the current mouse window. Note that there is no need to explicitly call this routine at the beginning of a program, since it will already have been called if a mouse is installed.

Example

```
SetFont8x8(True);
FullMouseWindow;
```

Switch into 43-/50-line mode and reset the mouse's window coordinates.

See Also

MouseWindow

Declaration

```
procedure GetMickeyCount(var Horizontal, Vertical : Integer);
```

Purpose

Return the horizontal and vertical mickey count since the last call to this function.

Description

This routine returns values indicating how far the mouse has been moved since the last time GetMickeyCount was called. Negative numbers indicate movement up (Vertical) or to the left (Horizontal); positive numbers indicate movement down (Vertical) or to the right (Horizontal).

GetMickeyCount is perhaps most useful in cases where you need to be able to determine whether or not the user has been moving the mouse, even if the reported position of the mouse cursor remains unchanged because it has reached the edge of the mouse window. For example, if the mouse cursor is located at row 1, column 1, and GetMickeyCount returns a Horizontal value of 0 and a Vertical value of -16, then you would know that user had been trying to move the mouse upward.

See the discussion of "The Mickey" at the beginning of the OPMOUSE section for an explanation of the term "mickey."

See Also

MouseWhereXY SetMickeyToPixelRatio

GetMousePage / HardMouseCursor

Declaration

```
function GetMousePage : Byte;
```

Purpose

Returns the video page where the mouse is being displayed.

Description

This routine returns the number of the video page where the mouse cursor is currently being displayed.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

Example

```
SaveMousePage := GetMousePage;
SetMousePage(NewMousePage);
...
SetMousePage(SaveMousePage);
```

This saves the current mouse page, switches to a new page, then restores the old one.

See Also

SetMousePage

Declaration

```
procedure HardMouseCursor(StartLine, EndLine : Word);
```

Purpose

Set mouse to use the hardware cursor.

Description

This routine does two things: it selects a hardware mouse cursor rather than a software cursor (the mouse driver's default choice), and it lets you specify the starting (StartLine) and ending (EndLine) scan lines for that cursor. This is a low-level routine, and in most cases you will probably prefer to use one of the higher level routines in OPMOUSE that call it: BlockMouseCursor, FatMouseCursor, HiddenMouseCursor, or NormalMouseCursor.

For more details, see the discussion of "The Mouse Cursor" at the beginning of the OPMOUSE section.

Example

```
if Font8x8Selected or (CurrentMode <> 7) then
  HardMouseCursor(0, 7)
else
  HardMouseCursor(0, 12);
```

Like the BlockMouseCursor routine, this would select a hardware cursor shaped like a solid block.

See Also

BlockMouseCursor
FatMouseCursor
HiddenMouseCursor
NormalMouseCursor
SoftMouseCursor

HiddenMouseCursor / HideMouse / HideMousePrim

Declaration
```
procedure HiddenMouseCursor;
```

Purpose
Set scan lines for a hidden mouse cursor.

Description
This routine selects a hardware mouse cursor that is (temporarily) hidden. It is analogous to the HiddenCursor routine in OPCRT. Note that this routine does not perform the same function as HideMouse, which can hide either a software or a hardware mouse cursor.

See Also
HardMouseCursor HiddenCursor (OPCRT)

Declaration
```
procedure HideMouse;
```

Purpose
Hide the mouse cursor.

Description
This routine hides the mouse cursor, no matter what its kind or shape. Probably the most common use for it is in situations where a software cursor is being used, and either the screen is about to be scrolled or a copy of the screen is about to be made (by a routine in OPWINDOW, perhaps). For more details, see the discussion of "The Mouse Cursor" at the beginning of the OPMOUSE section.

Example
```
HideMouse;
{write to the screen, scroll display, save an image of the display, whatever}
ShowMouse;
```

See Also
InitializeMouse ShowMouse

Declaration
```
procedure HideMousePrim(var MouseState : Boolean);
```

Purpose
Save state of mouse cursor in MouseState and hide it.

Description
This routine is used in conjunction with ShowMousePrim. It saves the current value of the global variable MouseCursorOn in MouseState, then hides the mouse cursor if necessary. A subsequent call to ShowMousePrim, using MouseState as a parameter, will restore the mouse cursor to the state (hidden or visible) it was in when HideMousePrim was called.

Example
```
var MouseState : Boolean
...
HideMousePrim(MouseState);
{write to the screen, scroll display, save the screen, whatever}
ShowMousePrim(MouseState);
```

Hide the mouse cursor, do something to the display, then restore the mouse cursor to the state it was in when HideMousePrim was called.

See Also
ShowMousePrim

InitializeMouse / KeyOrButtonPressed

Declaration

```
procedure InitializeMouse;
```

Purpose

Reinitializes mouse and sets MouseInstalled.

Description

This routine reinitializes the mouse and sets MouseInstalled. Although it is called automatically at the beginning of the program, there are cases where it might need to be called later as well–in a TSR, for example, which must deal with the possibility that the mouse driver has been installed/uninstalled since the TSR was first loaded. In "reinitializing" the mouse, it does the following:

1) Hides the mouse cursor. 2) Selects a software mouse cursor (still hidden) with a screen mask of $FFFF (DefaultScreenMask) and a cursor mask of $7700 (DefaultCursorMask). 3) Selects an 80 x 25 mouse window (40 x 25 in 40-column modes). 4) Moves the cursor to the center of that window. 5) Selects a mickey-to-pixel ratio of 8,16. And 6) disables any user-defined event handler.

Example

```
SaveMouseState(MSP, False);
InitializeMouse;
EnableEventHandling;
BlockMouseCursor;
ShowMouse;
...
RestoreMouseState(MSP, False);
```

A fairly typical series of statements in a TSR that uses the mouse: Save the state of the mouse driver, reinitialize the mouse, then select a hardware mouse cursor and make it visible. Later the previous state of the mouse driver is restored.

See Also

HideMouse ShowMouse

Declaration

```
function KeyOrButtonPressed : Boolean;
```

Purpose

Return True if a key or mouse button has been pressed.

Description

This routine simply calls KeyPressed and MousePressed and returns True if either is True.

See Also

MousePressed

Declaration

```
function MouseButtonPressed(Button : ButtonStatus; var Count : Word;
                            var LastX, LastY : Byte) : Boolean;
```

Purpose

Return True if the Button to check has been pressed.

Description

This routine allows you to determine whether or not a particular mouse Button has been pressed. If it has, Count returns the number of times it has been pressed since the last call to MouseButtonPressed, and LastX and LastY will contain the mouse cursor's position the last time it was pressed.

In general, we recommend that you use an event handler in cases where you are tempted to use this function and its companion, MouseButtonReleased. These two routines are provided primarily for completeness.

Example

```
if MouseButtonPressed(LeftButton, Count, LastX, LastY) then
  WriteLn('The left mouse button has been pressed ', Count, ' times);
```

Display the number of times that the left mouse button has been pressed.

See Also

MouseWhereXY

Declaration

```
function MouseButtonReleased(Button : ButtonStatus; var Count : Word;
                             var LastX, LastY : Byte) : Boolean;
```

Purpose

Return True if the Button to check has been released.

Description

This routine allows you to determine whether or not a particular mouse Button has been released. If it has, Count returns the number of times it has been released since the last call to MouseButtonReleased, and LastX and LastY will contain the mouse cursor's position the last time it was released.

Example

```
if MouseButtonReleased(LeftButton, Count, LastX, LastY) then
  WriteLn('The left mouse button has been released ', Count, ' times);
```

Display the number of times that the left mouse button has been released.

MouseGotoXY / MouseInWindow

Declaration

```
procedure MouseGotoXY(MouseX, MouseY : Byte);
```

Purpose

Set mouse position.

Description

This routine is analogous to the GotoXY routine in CRT and OPCRT. It moves the mouse cursor to the specified location. As elsewhere in OPMOUSE, the screen coordinates specified here are interpreted as being relative to the current mouse window. If the coordinates are outside the mouse window, MouseGotoXY does nothing.

Note that some older mouse drivers do not fully support this function, and will move the mouse cursor only if it is hidden. If your users run into this problem, they should contact the maker of the mouse driver about getting an update. The problem is in the mouse driver, not in MouseGotoXY.

Example

```
MouseWindow(5, 5, 75, 20);
MouseGotoXY(1, 1);
```

Moves the mouse cursor to row 5, column 5.

See Also

MouseWindow

Declaration

```
function MouseInWindow(XLo, YLo, XHi, YHi : Byte) : Boolean;
```

Purpose

Return True if the mouse cursor is within the specified window.

Description

This routine returns True if the mouse cursor is currently within the specified region of the screen. Note that the screen coordinates passed in as parameters are interpreted as being relative to the current mouse window.

Example

```
MouseWindow(5, 5, 75, 20);
...
if MouseInWindow(1, 1, 3, 3) then {do something};
```

This checks to see if the mouse cursor is somewhere in the first three rows and first three columns of the current mouse window.

Declaration

```
function MouseKeyWord : Word;
```

Purpose

Return a pseudo-scan code based on which mouse button is being pressed.

Description

This routine is analogous to the ReadKeyWord function in OPCRT. Assuming that EnableEventHandling has been called, it sits in a loop waiting for a mouse button to be pressed, then returns a pseudo-scan code in the range MouseLft..MouseThree that indicates which button or buttons were pressed. If EnableEventHandling has not been called, it will return $FFFF.

Once a mouse button has been pressed, MouseKeyWord does one of two things, depending on the current value of WaitForButtonRelease. If this variable is False, it will immediately return the pseudo-scan code for the button or combination of buttons that was being pressed. If it is True, MouseKeyWord will continue to monitor the status of the mouse buttons until they have all been released. In either case, on exit the routine will store the current location of the mouse cursor in MouseKeyWordX and MouseKeyWordY.

See the discussion on "Reading The Mouse Buttons" at the beginning of the OPMOUSE section for more details.

Example

```
ScanCode := $FFFF;
repeat
  if KeyPressed then
    ScanCode := ReadKeyWord
  else if MousePressed then
    ScanCode := MouseKeyWord;
until ScanCode <> $FFFF;
```

Loop until a key or a mouse button is pressed. (This is essentially what ReadKeyOrButton does.)

See Also

EnableEventHandling ReadKeyOrButton

Declaration

```
function MousePressed : Boolean;
```

Purpose

Return True if a mouse button is currently being pressed.

Description

This routine is analogous to the KeyPressed function in CRT and OPCRT. It tells you whether or not a mouse button is currently being pressed. Like MouseKeyWord, it does nothing useful if EnableEventHandling has not been called. (That is, it always returns False.) See the entry for MouseKeyWord for an example.

See Also

EnableEventHandling KeyOrButtonPressed

MouseStateBufferSize / MouseWhereX

Declaration

```
function MouseStateBufferSize : Word;
```

Purpose

Returns amount of memory needed to save the state of the mouse driver.

Description

This function returns the size of the buffer needed to save the state of the mouse driver. It is intended primarily for use in TSRs that need to provide mouse support. The value returned will vary from one mouse driver to another, but should be roughly in the range 350-400 (bytes). If a mouse is not installed, 0 will be returned.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. It is written in such a way that 0 should be returned if the function isn't implemented, but there's no guarantee that that will happen. You should therefore 1) check for a return value of 0, 2) check for unusually large values (higher than, say, 1000) and 3) warn users of the potential problem. If a problem does arise, they can probably correct it only by updating their mouse driver software.

For an example of how to use this routine, see the introductory section on "Using the Mouse in a TSR."

See Also

RestoreMouseState SaveMouseState

Declaration

```
function MouseWhereX : Byte;
```

Purpose

Return current X coordinate for mouse.

Description

This routine is analogous to the WhereX function in CRT and OPCRT. It returns the present X (column) coordinate for the mouse cursor, relative to the current mouse window. To obtain the absolute column coordinate of the mouse, call MouseWhereXabs instead.

See Also

MouseWhereXabs MouseWindow
MouseWhereY

Declaration

```
function MouseWhereXabs : Byte;
```

Purpose

Return current X coordinate (absolute) for mouse.

Description

This routine is analogous to the WhereXabs function in OPCRT. Unlike MouseWhereX, it returns the absolute X (column) coordinate for the mouse cursor.

See Also

MouseWhereX WhereXabs (OPCRT)

Declaration

```
procedure MouseWhereXY(var MouseX, MouseY : Byte; var Status : ButtonStatus);
```

Purpose

Return mouse position and button status.

Description

This routine returns the present location of the mouse cursor, relative to the current mouse window, in MouseX and MouseY. (To obtain the absolute row and column coordinates of the mouse, use MouseWhereXYabs instead.) The current state of the mouse buttons is returned in Status. This routine is especially useful in cases where you need to determine which buttons, if any, are being pressed, and execute different functions based on the location of the mouse cursor.

Example

```
MouseWhereXY(MouseX, MouseY, Status);
case Status of
  LeftButton  : HandleLeftButton;  {left button pressed}
  RightButton : HandleRightButton; {right button pressed}
  BothButtons : HandleBothButtons; {left and right buttons both pressed}
end;
```

Get the location of the mouse cursor and the status of the mouse buttons, then execute a particular procedure based on which combination of buttons has been pressed.

See Also

MouseWhereX MouseWindow
MouseWhereY

MouseWhereXYabs / MouseWhereY

Declaration

```
procedure MouseWhereXYabs(var MouseX, MouseY : Byte; var Status : ButtonStatus);
```

Purpose

Return mouse position (absolute) and button status.

Description

This routine is identical to MouseWhereXY except that the coordinates it returns in MouseX and MouseY are absolute, not relative to the current mouse window.

Example

See the example for MouseWhereXY.

See Also

MouseWhereXabs
MouseWhereXY
MouseWhereYabs

Declaration

```
function MouseWhereY : Byte;
```

Purpose

Return current Y coordinate for mouse.

Description

This routine is analogous to the WhereY function in CRT and OPCRT. It returns the present Y (row) coordinate for the mouse cursor, relative to the current mouse window. To obtain the absolute row coordinate of the mouse, add MouseYLo to the value returned by MouseWhereY.

See Also

MouseWhereX
MouseWhereXY
MouseWhereYabs

Declaration

```
function MouseWhereYabs : Byte;
```

Purpose

Return current Y coordinate (absolute) for mouse.

Description

This routine is analogous to the WhereYabs function in OPCRT. Unlike MouseWhereY, it returns the absolute Y (row) coordinate for the mouse cursor.

See Also

MouseWhereY WhereYabs (OPCRT)

Declaration

```
procedure MouseWindow(XLow, YLow, XHigh, YHigh : Byte);
```

Purpose

Sets window coordinates to be observed by the mouse.

Description

This routine is similar to the Window procedure in CRT and OPCRT. It sets the minimum/maximum horizontal and vertical positions on the screen for the mouse cursor. The screen coordinates are subjected to the same kind of scrutiny as those passed to Window: XLow/YLow cannot be greater than XHigh/YHigh, XHigh cannot be greater than ScreenWidth, and YHigh cannot be greater than ScreenHeight. If any of the coordinates is invalid, MouseWindow does nothing. If the coordinates are valid, however, the mouse cursor will be moved inside the window if isn't already, and the new window coordinates will be stored in MouseXLo, MouseXHi, MouseYLo, MouseYHi.

The default mouse window is the entire screen, based on the width and height of the screen when the program began. If the dimensions of the screen change later on, you should call FullMouseWindow to reset the mouse's window coordinates.

Example

```
Window(1, 1, 80, 25);
if MouseInstalled then
  MouseWindow(1, 1, 80, 25);
```

Create an 80 x 25 window and a mouse window to match. Notice that MouseWindow is not called unless MouseInstalled is True.

See Also

FullMouseWindow

NormalMouseCursor / ReadKeyOrButton

Declaration

```
procedure NormalMouseCursor;
```

Purpose

Set normal scan lines for mouse cursor based on current video mode.

Description

This routine selects a hardware mouse cursor that is two-lines high. It is analogous to the NormalCursor routine in OPCRT.

See Also

HardMouseCursor NormalCursor (OPCRT)

Declaration

```
function ReadKeyOrButton : Word;
```

Purpose

Return code for next key or mouse button pressed.

Description

This routine is provided partly for internal use, and partly as a convenience. It simply sits in a loop waiting for a key or mouse button to be pressed, then returns either the scan code for the key, if a key was pressed first, or the pseudo-scan code for the mouse button, if a button was pressed first. Assuming that a mouse button was pressed first, on exit the current location of the mouse cursor will be stored in MouseKeyWordX and MouseKeyWordY.

ReadKeyOrButton uses MousePressed and MouseKeyWord to obtain information about the mouse buttons, so it is subject to the same limitation that they are: EnableEventHandling must be called before it can obtain any information about the mouse buttons.

See the discussion on "Reading The Mouse Buttons" at the beginning of the OPMOUSE section for more details.

Example

```
var
  I : Word;
...
  I := ReadKeyOrButton;
  if Lo(I) = 0 then
    case I of
      $0000 : {^Break was pressed} ;
      ...
      MouseLft : {left mouse button was pressed} ;
    end;
```

Get the next key or mouse button press and process the command.

See Also

EnableEventHandling (OPCRT) ReadKeyWord (OPCRT)
MouseKeyWord (OPCRT)

Declaration

```
procedure RestoreMouseCoordinates(WC : WindowCoordinates);
```

Purpose

Restore previously saved mouse window coordinates.

Description

This routine, analogous to the RestoreWindowCoordinates routine in OPCRT, is intended to be used in conjunction with StoreMouseCoordinates in cases where mouse window coordinates need to be changed temporarily and later restored. WC, a record variable of type WindowCoordinates (declared in OPCRT), contains the four coordinates for the mouse window.

Example

```
var
  WC : WindowCoordinates;
...
  StoreMouseCoordinates(WC);
  FullMouseWindow;
  ...
  RestoreMouseCoordinates(WC);
```

Save the current mouse window coordinates, reset the mouse window, then restore the original coordinates.

See Also

RestoreWindowCoordinates (OPCRT) StoreMouseCoordinates

Declaration

```
procedure RestoreMouseState(var MSP : MouseStatePtr; Deallocate : Boolean);
```

Purpose

Restore the state of the mouse driver and Deallocate the buffer if requested.

Description

This function restores the state of the mouse driver. It is intended primarily for use in TSRs that need to provide mouse support. On entry, MSP points to the buffer containing the data to be restored. If Deallocate is True, this buffer will be deallocated on exit and MSP will be set to nil. If a mouse is not installed or MSP is nil on entry, RestoreMouseState will do nothing.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

For an example of how to use this routine, see the introductory section on "Using the Mouse in a TSR."

SaveMouseState / SetMickeyToPixelRatio

Declaration

```
procedure SaveMouseState(var MSP : MouseStatePtr; Allocate : Boolean);
```

Purpose

Save the state of the mouse driver, allocating the buffer if requested.

Description

This function saves the state of the mouse driver. It is intended primarily for use in TSRs that need to provide mouse support. MSP points to the buffer used to save the mouse driver's data. If Allocate is True, this buffer will be allocated on the heap and MSP will point to it on exit. If it is False, MSP should point on entry to a previously allocated buffer. If a mouse is not installed, or MouseStateBufferSize returns 0 when asked for the size of the buffer needed, MSP will be set to nil.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

For an example of how to use this routine, see the introductory section on "Using the Mouse in a TSR."

See Also

MouseStateBufferSize

Declaration

```
procedure SetMickeyToPixelRatio(Horizontal, Vertical : Integer);
```

Purpose

Sets the mickey-to-pixel ratio.

Description

For an explanation of the concepts of the "mickey" and the "mickey-to-pixel ratio", see the discussion of "The Mickey" at the beginning of the OPMOUSE section.

This routine lets you alter the mickey-to-pixel ratio(s). Although the default settings of 8,16 (Horizontal,Vertical) are usually appropriate, there may be cases where you want to increase or decrease the mouse's sensitivity to movement. A setting of 16,32 will "slow down" the mouse considerably by making it less sensitive to movement, while a setting of 4,8 will make it "move faster" by increasing its sensitivity to movement.

The Horizontal and Vertical parameters should both be integers in the range 1 to 32,767 (MaxInt).

Examples

```
SetMickeyToPixelRatio(4,8);
```

Speed things up.

```
SetMickeyToPixelRatio(16,32);
```

Slow things down.

Declaration

```
procedure SetMouseEventHandler(EventMask : MouseEventType;
                               UserRoutine : Pointer);
```

Purpose

Sets the address of a routine to be called when the specified mouse events occur.

Description

Unlike the frequently-mentioned EnableEventHandling routine, the main purpose of this routine is not to activate OPMOUSE's own event handler (although that will be done if necessary), but rather to specify a user-written routine (pointed to by UserRoutine) that it is to call when selected mouse events occur.

If not nil, UserRoutine should point to a FAR-declared procedure that takes no parameters and is global in scope (that is, it may not be nested inside another procedure). Information about the Event that just took place may be obtained by checking the global variables MouseEvent, MouseStatus, MouseLastX, and MouseLastY. If UserRoutine is nil (or EventMask = DisableEventHandler), then any user-defined event handler currently in place will be "disabled" (not called).

EventMask is a bit-mapped byte that tells OPMOUSE's event handler when your routine should be called. For example, if you want it to be called only when the left button is pressed or released, the EventMask should be 'LeftButtonPressed+LeftButtonReleased'. If you wanted to know about all mouse events, you would use AllMouseEvents instead.

Note that OPMOUSE handles all the "dirty work" involved in writing an event handler for you: it saves all the CPU registers, sets DS to point to your program's own data segment, calls UserRoutine, then restores the CPU registers when it regains control.

For more details, see the discussions of "Mouse Event Handlers" at the beginning of the OPMOUSE section.

See Also

DisableEventHandling EnableEventHandling

Declaration

```
procedure SetMousePage(Page : Byte);
```

Purpose

Sets the video page where the mouse will be displayed.

Description

This routine, analogous to the SetVisiblePage routine in OPCRT, tells the mouse driver which video page to use when displaying the mouse cursor.

Warning: This routine uses a mouse function that is not implemented in all mouse drivers. See the discussion of MouseStateBufferSize for more details.

Example

```
SetVisiblePage(3);
SetMousePage(CurrentPage);
```

Switch both the display and the mouse cursor to video page 3. Notice that SetMousePage uses CurrentPage, in case the call to SetVisiblePage failed.

See Also

GetMousePage SetVisiblePage (OPCRT)

ShowMouse / ShowMousePrim

Declaration

```
procedure ShowMouse;
```

Purpose

Show the mouse cursor.

Description

This routine must be called in order to reveal the mouse cursor, whether it is a software cursor or a hardware cursor. For more details, see the discussion of "The Mouse Cursor" at the beginning of the OPMOUSE section.

Example

```
BlockMouseCursor;
ShowMouse;
...
HideMouse;
```

Select a hardware mouse cursor shaped like a block, unhide the cursor, do something, then re-hide the cursor.

See Also

HideMouse InitMouse

Declaration

```
procedure ShowMousePrim(MouseState : Boolean);
```

Purpose

Hide or unhide the mouse cursor.

Description

This routine is intended to be used in conjunction with HideMousePrim. What it does, when used that way, is to restore the mouse state (hidden or visible) to what it was when HideMousePrim was called. More generally, it will make the mouse cursor visible ("show" it) if MouseState is True, or hide it if MouseState is False.

Example

See the example for HideMousePrim.

Declaration

```
procedure SoftMouseCursor(ScreenMask, CursorMask : Word);
```

Purpose

Set mouse to use a software cursor.

Description

This routine does two things: 1) it selects a software cursor (the default) rather than a hardware cursor, and 2) it lets you specify, in a somewhat roundabout fashion, the shape and color of the cursor.

For details on the subject of mouse cursors, including screen masks and cursor masks, see the discussion of "The Mouse Cursor" at the beginning of the OPMOUSE section.

Examples

```
SoftMouseCursor(DefaultScreenMask, DefaultCursorMask);
```

Selects the default type of software cursor, a non-blinking block whose color varies depending on where the cursor is located.

```
SoftMouseCursor($00FF, $F000);
```

Selects a blinking reverse video (black on white) block cursor.

```
SoftMouseCursor($0000, $1F04);
```

Selects a white diamond (ASCII 4) on a blue background for a cursor.

See Also

HardMouseCursor

Declaration

```
procedure StoreMouseCoordinates(var WC : WindowCoordinates);
```

Purpose

Store the mouse window coordinates in WC.

Description

This routine, analogous to the StoreWindowCoordinates routine in OPCRT, is intended to be used in conjunction with RestoreMouseCoordinates in cases where mouse window coordinates need to be changed temporarily and later restored. On exit, WC, a record variable of type WindowCoordinates (declared in OPCRT), will contain the current coordinates for the mouse window.

Example

See the example for RestoreMouseCoordinates.

See Also

RestoreMouseCoordinates StoreWindowCoordinates (OPCRT)

E. OPCMD: Command Processing

One of the most basic tasks performed by any interactive program is the translation of keystrokes into commands. In its simplest form, the process is straightforward, something like this perhaps:

```
Ch := ReadKey;
if Ch = #0 then begin
  Ch := ReadKey;
  case Ch of
    #$3B : {F1} ;
    #$3C : {F2} ;
    else   {something else} ;
  end;
end
else case Ch of
  #27 : {Escape} ;
  ^H  : {Backspace} ;
  ^I  : {Tab} ;
  else  {something else} ;
end;
```

There's little involved in the process: we look at the key that was pressed, and we execute the appropriate section of code based on the "command" that the key represents. As we all know, this approach works perfectly well in many situations. But in others it is completely inadequate.

Suppose, for example, that you use this approach throughout your program, and then your client or your users clamor for a way to change the key assignments, the correspondences between keystrokes and "commands." Obviously, you just can't do it without making numerous changes to your program, because all of the key assignments are hard-coded in the source. In order to fill the request, you would need to take an entirely different approach to the translation of commands.

What you would need, first of all, is a set of "command codes," numbers that represent all of the commands in the various parts of your programs. This would make possible the next thing you'd need, a set of tables that show the relationships between key sequences and commands in the various parts of your programs. Each table would contain a series of entries consisting of a command code and one or more key codes that go with it.

The next step would be to write a routine to perform the actual translation of commands at runtime. Given a keystroke, or series of keystrokes, it would scan the appropriate table and return the code for the corresponding command. The caller of this routine wouldn't have to know anything about the actual key that was pressed, just the command that it represents.

Having moved all the key assignments into data structures of a known format, it would then become feasible to give the user a way to change the assignments, perhaps via a configuration routine in the same program, or perhaps by means of a separate installation program that locates the tables in the main program's EXE file, modifies them, then puts them back.

Once you had made the change, other benefits would also become apparent. For example, if the same table was used in three different parts of the program, you would be able to change the key assignments for all three simply by changing the one table.

As you've surely guessed by now, what we've been describing here is the approach to command translation implemented throughout **Object Professional** using the facilities provided in OPCMD.

1. Command Codes

For starters, OPCMD defines a series of standard command codes. These codes are all simple constants of the form 'ccXxx = n', where 'cc' identifies the constant as a command code, 'Xxx' is the name of the command, and 'n' is its pre-assigned value. A complete list of command codes appears later in this chapter, along with a brief note on what they mean.

A few of the command codes are widely used. ccQuit, for example, means "Quit processing" and, possibly, "Ignore changes". It is generally associated with the <Esc> key. ccLeft, ccRight, ccUp, and ccDown mean "Move the cursor in the specified direction," and they usually correspond to the <Left>, <Right>, <Up> and <Down> keys. These generic commands have fairly widespread applications: in menus, pick lists, line editors, text editors, etc. Other commands in the list are used in only one or two cases. ccBlkToggle, for example, which means "Toggle the display of marked blocks," is used only by file browsers (OPBROWSE) and text editors (OPEDITOR).

But the single largest group of command codes are those set aside for "user exit commands": ccUser0...ccUser54. Collectively, these commands represent one of the most useful hooks provided by **Object Professional**, one that allows you to designate key sequences that have are to have special meaning for a given program. For example, let's say that you want <CtrlF2> to mean "Save everything and exit from the program" when the user is at your main menu. There's no standard command that has that meaning, so what you would do is modify the command table for the menu system so that <CtrlF2> is assigned to one of the user exit commands, say ccUser0. When the menu system sees that a user exit command has been issued, it would return control to you and allow you to process the command. How do you change the table? We'll cover that in section 2, concerning the CommandProcessor object.

Declarations

```
ccNone        = 000; {Invalid command}
ccChar        = 001; {A regular character (alphanumeric, punctuation, etc.)}
ccCtrlChar    = 002; {Accept control character}
ccSelect      = 003; {Select/accept item, field; new line}
ccQuit        = 004; {Quit processing, ignore changes}
ccError       = 005; {Quit due to "fatal" error}
ccMouseSel    = 006; {Mouse selection of item, field, etc.}
ccRestore     = 007; {Restore default value and continue}
ccHome        = 008; {Cursor to beginning of line/list}
ccEnd         = 009; {Cursor to end of line/list}
ccLeft        = 010; {Cursor left by one character/column}
ccRight       = 011; {Cursor right by one character/column}
ccUp          = 012; {Up}
ccDown        = 013; {Down}
ccScrollUp    = 014; {Scroll window up}
ccScrollDn    = 015; {Scroll window down}
ccWordLeft    = 016; {Cursor left one word}
ccWordRight   = 017; {Cursor right one word}
ccPageUp      = 018; {Page up}
ccPageDn      = 019; {Page down}
ccScreenTop   = 020; {Top of screen}
ccScreenBot   = 021; {Bottom of screen}
ccTopOfFile   = 022; {Top of file}
ccEndOfFile   = 023; {End of file}
ccBack        = 024; {Backspace one character}
ccDel         = 025; {Delete current character}
ccDelEol      = 026; {Delete from cursor to end of line}
```

```
    ccDelBol      = 027; {Delete from beginning of line to cursor}
    ccDelLine     = 028; {Delete entire line}
    ccDelWord     = 029; {Delete word to right of cursor}
    ccIns         = 030; {Toggle insert mode}
    ccHelp        = 031; {Invoke help routine}
    ccTab         = 032; {Cursor to next tab stop/subfield}
    ccBackTab     = 033; {Cursor to previous tab stop/subfield}
    {Special OPMEMO/OPEDITOR commands}
    ccIndent      = 034; {Toggle indent mode}
    ccWordWrap    = 035; {Toggle word wrap}
    ccReformatP   = 036; {Reformat paragraph}
    ccReformatG   = 037; {Global reformat}
    {Special OPSELECT/OPENTRY commands}
    ccIncChoice   = 038; {Increment choice}
    ccDecChoice   = 039; {Decrement choice}
    ccNextField   = 040; {Cursor to next field}
    ccPrevField   = 041; {Cursor to previous field}
    ccNextRec     = 042; {Next record (an exit command)}
    ccPrevRec     = 043; {Previous record (an exit command)}
    ccFirstFld    = 044; {Cursor to first field}
    ccLastFld     = 045; {Cursor to last field}
    ccNested      = 046; {Exit for nested form}
    ccDone        = 047; {Completely finished with editing}
    {Special commands generated internally--should not be assigned to keys}
    ccClickExit   = 048; {Double mouse click produced an exit}
    ccAutoAdvance = 049; {Auto-advance to next field}
    ccExitAtTop   = 050; {Exit at top edge of window}
    ccExitAtBot   = 051; {Exit at bottom edge of window}
    ccExitLeft    = 052; {Exit at left edge of window}
    ccExitRight   = 053; {Exit at right edge of window}
    {Special OPHELP commands}
    ccPrevTopic   = 055; {Display the most recent help topic}
    ccIndex       = 056; {Display an index of all help topics}
    ccNextXref    = 057; {Jump to the next cross-reference}
    ccPrevXref    = 058; {Jump to the previous cross-reference}
    {Special OPBROWSE/OPEDITOR commands}
    ccSetMark0    = 060; {Set text marker 0..9}
    {ccSetMark1...ccSetMark8}
    ccSetMark9    = 069;
    ccJmpMark0    = 070; {Jump to text marker 0..9}
    {ccJmpMark1...ccJmpMark8}
    ccJmpMark9    = 079;
    ccMarkToggle  = 080; {Toggle text marker display}
    ccBlkBegin    = 081; {Mark start of block}
    ccBlkEnd      = 082; {Mark end of block}
    ccBlkBottom   = 083; {Mark end of block (at bottom of browse window)}
    ccBlkToggle   = 084; {Toggle block display}
    ccBlkRead     = 085; {Read block}
    ccBlkWrite    = 086; {Write block}
    ccBlkCopy     = 087; {Copy block}
    ccBlkMove     = 088; {Move block}
    ccBlkDelete   = 089; {Delete block}
    ccBlkWord     = 090; {Mark current word as block}
    ccBlkUCase    = 091; {Change characters in block to upper case}
    ccBlkLCase    = 092; {Change characters in block to lower case}
    ccBlkTCase    = 093; {Toggle case of characters in block}
    ccBlkIndent   = 094; {Indent marked block one level}
    ccBlkUnindent = 095; {Unindent marked block one level}
    ccSetIndent   = 096; {Set indentation level for block indent/unindent}
```

```
  ccPrevPos      = 097; {Cursor to previous position in file}
  ccJmpBegin     = 098; {Jump to beginning of marked block}
  ccJmpEnd       = 099; {Jump to end of marked block}
  ccJmpLine      = 100; {Jump to line}
  ccInsertLine   = 101; {Insert line at cursor}
  ccCenterLine   = 102; {Center current line}
  ccSearch       = 103; {Search}
  ccReplace      = 104; {Search and replace}
  ccReSearch     = 105; {Search again}
  ccHexMode      = 106; {Toggle hex mode}
  ccTabExpand    = 107; {Toggle tab expansion}
  ccTabToggle    = 108; {Toggle smart tabs/fixed tabs}
  ccTabSize      = 109; {Set size of fixed tabs}
  ccRtMargin     = 110; {Set right margin}
  ccStripHigh    = 111; {Toggle stripping of high bits}
  ccNewFile      = 112; {Read a new file}
  ccSaveFile     = 113; {Save a file}
  ccSaveNamed    = 114; {Save a file under a new name}
  ccSaveExit     = 115; {Save file and exit}
  ccSaveSwitch   = 116; {Save file and load new one}
  ccAbandonFile  = 117; {Abandon file}
  ccBlkPrint     = 118; {Print marked block}
  {Special OPPICK multi-item selection commands}
  ccToggle       = 119; {Toggle selection of an item}
  {Special OPMENU submenu selection commands}
  ccSubPrev      = 120; {Move to previous submenu}
  ccSubNext      = 121; {Move to next submenu}
  ccSubQuit      = 122; {Exit current submenu}
  {Diagonal cursor movement commands}
  ccUpLeft       = 130; {Cursor up and to the left}
  ccDownLeft     = 131; {Cursor down and to the left}
  ccUpRight      = 132; {Cursor up and to the right}
  ccDownRight    = 133; {Cursor down and to the right}
  {User-defined exit commands}
  ccUser0        = 200;
  {ccUser1...ccUser53}
  ccUser54       = 254;
  ccUser55       = 255; {Reserved for internal use}
  ccUser65335    = 65535; {Values > ccUser55 valid only if using word commands}
```

The standard command codes in **Object Professional**. Most of the commands in the first group (ccNone..ccBackTab) are widely used. Commands in later groups, excluding the last, are typically used by only one or two objects. The final group, the user exit commands, are recognized by all objects. Note that there are no named commands between ccUser55 and ccUser65335, which simply names the highest possible value for a user exit command.

2. The CommandProcessor

The most important facility provided by OPCMD is an object called the CommandProcessor, whose main job is to translate key sequences into command codes. Probably the best way to understand how it works is to study a simple example, such as the one shown below.

```
program CommandProcessorExample; {EXCMDPRC.PAS}
uses
  OpCrt, OpCmd;
const
  {Define a small KeySet of a few cursor commands}
  KeySetId : string[14] = 'KeySet follows';
  KeyMax  = 30;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys          command type    key sequence}
  3,      $00, $48,     ccUp,           {Up}
  3,      $00, $50,     ccDown,         {Down}
  3,      $00, $4B,     ccLeft,         {Left}
  3,      $00, $4D,     ccRight,        {Right}
  2,      $1B,          ccQuit,         {Esc}
  0,0,0,0,0,0,0,0,0,0,0,0);             {<-room for more commands}
var
  Commands : CommandProcessor;
  Cmd, Key : Word;
begin
  {Make a small CommandProcessor}
  Commands.Init(@KeySet, KeyMax);

  {Add a user exit command -- <F1>}
  Commands.AddCommand(ccUser0, 1, $3B00, 0);
  {check for error adding the command}
  if Commands.GetLastError <> 0 then
    RingBell;

  repeat
    Cmd := Commands.GetCommand(Key);
    case Cmd of
      ccChar  : WriteLn('ccChar: "', Char(Lo(Key)), '"' );
      ccUp    : WriteLn('ccUp');
      ccDown  : WriteLn('ccDown');
      ccLeft  : WriteLn('ccLeft');
      ccRight : WriteLn('ccRight');
      ccQuit  : WriteLn('ccQuit');
      ccUser0 : WriteLn('ccUser0');
      else      WriteLn('ccNone');
    end;
  until (Cmd = ccQuit);
end.
```

Notice first the typed constant called KeySet, a 0-based array of Byte. This table contains all the key assignments for the program. As you can see, the information is packed into the array in a series of variably sized records, each containing three fields. The first field is always a length byte indicating the combined size of the next two fields. The second contains the keycode(s). The third contains the command code.

The second field is the one that varies in size. In most cases, commands will be assigned to a single key like <CtrlA> or <Home>, or to a two-key combination like <CtrlQ><CtrlD>, and the keycode will occupy either one or two bytes. Control characters require one byte each,

while keys with extended scan codes (function keys, cursor keys, etc.) require two bytes: a 0 followed by the scan code. Hence the size of the keycodes field will usually be between 1 byte (a single control character) and 4 bytes (a sequence of two keys with extended scan codes, such as <AltU><AltU>). Although it is possible to assign combinations of three or more keys to a single command, doing so is inadvisable.

At the end of the command table are a series of 0's. Except for the last one, these mean nothing. They simply set aside space for additional commands to be added later. The last one is important, however. When the table is being scanned for a match between a key entered by the user and a command, the search will continue, record by record, until a matching command is found or a length byte of 0 is encountered. Hence, there must always be at least one 0 at the end of table to mark the end of it.

On to the CommandProcessor itself. Commands is intialized by calling the Init constructor (which cannot fail) with the address of the table containing the key assignments, and a second parameter that tells it how large the table is. Notice that the value passed is not the size of the array, 'SizeOf(KeySet)', but rather the last valid index into the array. This is important. The CommandProcessor always assumes that the table is an 'array[0..n] of Byte', and that the value passed as the second parameter to Init represents n.

One thing that you can't see here is that Init initializes a group of procedure pointers stored in the CommandProcessor, the most important of which is the GetKeyProc. This function returns the next keystroke entered by the user. We'll talk more about this subject later. The relevant point here is that a CommandProcessor is itself capable of obtaining keystrokes from the user. You don't ask it to "translate this key into a command." Rather, you ask it to "get the next command."

The call to AddCommand adds a new key assignment, associating <F1> with the user exit command ccUser0. In this case, the operation is sure to succeed, but the call to GetLastError checks to be sure. If GetLastError returns any value other than 0, a beep will be sounded to indicate that the command was not added successfully.

The repeat loop then demonstrates the method used to obtain commands from the user through the intermediary of the CommandProcessor. The function result returned by GetCommand is the command that was issued, and the Key parameter returns the last key that was pressed. Notice in particular that, if a "regular" key was pressed–an alphanumeric character, a punctuation mark, or a high-order graphics character–and that character is not assigned to a specific command, then GetCommand will return 'ccChar'. This behavior allows line editors and text editors, for example, to distinguish between editing commands and ordinary text. In all other cases–control characters, function keys, cursor keys, etc.–'ccNone' will be returned if the key sequence does not correspond to a command. (ccChar will also be returned for unassigned control characters, #0..#31, if the cpCtrlsAreChars option is on.)

Word-sized Commands

If you look back at the above discussion of the format of a command table you'll notice that the command codes are represented as a single byte, limiting you to 256 commands for any one CommandProcessor. In the majority of applications, 256 will be more than enough. But in rare cases–a sophisticated programmer's editor or word processor, for example–you may need more. In these situations, you need to do only two things to get around the 256-command limit inherent in using bytes to represent command codes.

First, you need to activate the cpWordCmds option, like so:

```
Commands.cpOptionsOn(cpWordCmds);
```

Second, you need to modify your command table(s) such that the command codes are represented as words rather than bytes. For example, we would have to change the command table in our demo program as follows:

```
const
  KeyMax  = 35;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys          command type   key sequence}
  4,      $00, $48,     ccUp,     0,   {Up}
  4,      $00, $50,     ccDown,   0,   {Down}
  4,      $00, $4B,     ccLeft,   0,   {Left}
  4,      $00, $4D,     ccRight,  0,   {Right}
  3,      $1B,          ccQuit,   0,   {Esc}
  0,0,0,0,0,0,0,0,0,0,0,0);            {<-room for more commands}
```

We did three things here. First, we increased KeyMax from 30 to 35. Second, we added 1 to each of the length bytes. And third, we added an extra 0 to the end of each record, because each of the five command codes has a value < = 255. If we had a command code greater than 255, such as 'ccSpecialCmd = 300', the record would look something like this instead:

```
3,  $0D,  Lo(ccSpecialCmd), Hi(ccSpecialCmd),   {Enter}
```

The important things to remember are that low byte of the command code comes first, the high byte second, and that all the length bytes need to be adjusted if you switch from byte-sized commands to word-sized commands.

Mouse Support

One important feature of the CommandProcessor that was not demonstrated in our sample program is mouse support. All of the higher level screen objects in **Object Professional** support the mouse, and that support starts with their CommandProcessors.

As explained in the documentation for OPMOUSE, the main routines for reading the mouse buttons (ReadKeyMouse and ReadKeyOrButton) return "pseudo scan codes," word-sized values that look like extended scan codes. Among other things, this makes it feasible for a CommandProcessor to treat button presses the same as ordinary keystrokes. So the first thing you need to do to add mouse support to code that uses a CommandProcessor is to add mouse button codes to your command table. For example, the statement

```
Commands.AddCommand(ccQuit, 1, MouseRt, 0);
```

would make pressing the right mouse button equivalent to pressing <Esc>.

The second thing you need to do is inform the CommandProcessor that mouse support is desired, like so:

```
Commands.cpOptionsOn(cpEnableMouse);
```

Activating the cpEnableMouse option does three things: 1) it enables OPMOUSE's mouse event handler; 2) it sets the GetKeyProc to point to ReadKeyOrButton rather than ReadKeyWord; and 3) it sets the KeyPressedProc to point to KeyOrButtonPressed rather than KeyPressed. These three steps insure that the CommandProcessor can read the mouse buttons as well as the keyboard.

There are, of course, other things that you have to do to add mouse support to your program--you have to select a mouse cursor, turn it on, hide/restore it when writing text to the screen--but these are the only ones that the CommandProcessor cares about.

Background Task Hook

In the previous sections we mentioned something about a "GetKeyProc" and a "KeyPressedProc". These are procedure pointers stored in the CommandProcessor that tell it what routines to call to get the next key/button press and to determine if a key/button is waiting to be processed. Normally these point to ReadKeyWord and KeyPressed, if mouse support is disabled, or ReadKeyOrButton and KeyOrButtonPressed if it is enabled.

You can, however, point them to your own routines if you want, like this:

```
{$F+}
function MyKeyPressed : Boolean;
begin
  if KeyOrButtonPressed then
    MyKeyPressed := True
  else begin
    inline($cd/$28); {allow TSRs to pop up}
    {** perform background task here **}
    MyKeyPressed := KeyOrButtonPressed;
  end;
end;

function MyReadKey : Word;
begin
  while not MyKeyPressed do {wait} ;
  MyReadKey := ReadKeyOrButton;
end;
{$F-}
...
Commands.cpOptionsOn(cpEnableMouse);
Commands.SetGetKeyProc(MyReadKey);
Commands.SetKeyPressedProc(MyKeyPressed);
```

The goal here is to allow the time normally spent waiting for keystrokes to be used for other purposes–displaying the current time on the screen, perhaps.

It looks simple, and it is, but a few important points need to be made here.

First, the GetKeyProc and KeyPressedProc must have the same form as the functions shown above. They must be declared FAR using {$F+}, they must not be nested inside any other procedure, and they must return results of the specified type.

Second, at least one check for a key or button press should be made before performing any background task. This insures that there are no unnecessary delays in responding to keyboard input.

Third, the "inline($cd/$28)" statement is sometimes necessary to allow TSRs to pop up while you are waiting for keyboard input. It generates an interrupt $28, which is a universal signal to TSRs that it is safe to pop up. We recommend that you always include such a statement in your KeyPressedProc (if you have one) or your GetKeyProc (if you don't) so that you won't forget it in cases where it is needed. (TSRs are the most common cases.)

Finally, notice that mouse support is enabled *before* the two procedure pointers are reset. This is important, because 'cpOptionsOn(cpEnableMouse)' also resets the two pointers, so if that statement came last it would undo the work of the previous two.

Context Sensitive Help Hook

A similar facility provided by the CommandProcessor is the context sensitive help hook. This hook consists simply of a pointer to a procedure, known as a GetHelpProc, that is to be called each time the ccHelp command is issued by the user. When that command is issued, the procedure or object using the CommandProcessor will call its cpGetHelp method with three parameters that identify the nature of the request. If a GetHelpProc has been specified, those parameters will be passed on to it unchanged by the CommandProcessor.

A sample GetHelpProc is shown below. Note that it must be compiled FAR using {$F+} and that it may not be nested inside another procedure.

```
{$F+}
procedure MyGetHelpProc(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
begin
  {** display help **}
end;
{$F-}
...
Commands.SetHelpProc(MyGetHelpProc);
```

The first parameter is a "unit code," a value that identifies the source of the request. For example, if the help command is issued inside a menu, the unit code would be 'ucMenu'. A complete list of the standard unit codes may be found in the documentation for OPROOT. The second parameter is usually, but not necessarily, a pointer to the object using the CommandProcessor.

The third parameter is a value that indicates more precisely where the user was in the program when he asked for help. In the case of a data entry screen, for example, the help index identifies the field in the entry screen; in the case of a menu or pick list, the item that is currently highlighted.

The values passed as the last two parameters vary widely depending on the source of the help request. Please consult the documentation for the object/routine using the given CommandProcessor for a description of what is passed in a given situation.

For an elaborate example of a GetHelpProc, written with the aid of the OPHELP unit, see ENTRY.PAS.

Secondary Command Tables

In most cases, a single command table will be sufficient for each object or routine that uses a given CommandProcessor. In other cases, however, it is desirable to have a second command table that establishes a set of standard user exit commands, or "hot keys." A prime example of such a case is the demonstration program DESKPOP, where each window on the desktop must respond to requests to change windows, move/zoom/resize a window, invoke the local or main menu, etc. Although it would be possible to add these commands to all of the individual command tables, it is far more efficient to define a single command table containing all of the hot key assignments and ask the CommandProcessor to scan both command tables when a key is pressed.

The CommandProcessor therefore provides the SetSecondaryKeyPtr method, which allows you to specify a secondary command table, and the cpSwitchPriority option, which allows you to control the order in which the command tables are scanned. If the cpSwitchPriority option is in effect, the secondary command table is scanned first, then the primary command table. If there are no conflicts between the two tables, setting this option accomplishes nothing. It is

meaningful only in situations where a key sequence is assigned to one command in one of the tables and another command in the other.

See TPUI.PAS and TPUI.ICD for an example of how to use a secondary command table.

Auto-Execution of Commands

Another useful feature of the CommandProcessor is the SetCommandList method, which allows you to designate a list of commands that are to be executed automatically. Normally such a list would take the form of a simple array of bytes (if byte-sized commands are in use) or an array of words (if word-sized commands are in use). But SetCommandList doesn't care what data structure is actually used, so long as it may be *treated as* an array of bytes/words. For example, the following statements accomplish the same thing (these commands might be passed to a file browser created with OPBROWSE):

```
type
  CmdRec = record
             Cmd1, Cmd2, Cmd3 : Byte;
           end;
const
  CmdList1 : array[1..3] of Byte = (ccTopOfFile, ccHexMode, ccQuit);
  CmdList2 : CmdRec = (
               Cmd1 : ccTopOfFile;
               Cmd2 : ccHexMode;
               Cmd3 : ccQuit );
...
BrowseCommands.SetCommandList(@CmdList1, 3);
BrowseCommands.SetCommandList(@CmdList2, 3);
```

In each case, the first command moves the cursor to the top of the file being browsed, the second switches in or out of hex mode, and the third returns control to the caller.

The command list may also contain ordinary text, delimited on each side by the ccChar command, allowing automated responses to requests for user input. This facility might be used, for example, to force a TextEditor or Browser to search for a particular string. The general format of such a series of instructions would be

```
command, command, ... ccChar, --text--, ccChar, command, command
```

How does this work? At each point where the TextEditor or Browser would normally call a user-written routine to prompt for a string or character, it also calls the CommandProcessor's CommandStringPending method to see if the next command in the list of automated responses is ccChar. If so, it would call the GetCommandString to retrieve the string ('--text--') surrounded by the two ccChar commands.

Note that the CommandWindow object defined in OPWINDOW simplifies the process of using this facility. For details, see the documentation for the ProcessAutoPilot method.

For examples of how commands can be auto-executed, see the source code for the demonstration program DESKPOP. DESKPOP uses this facility fairly extensively to allow various commands to be executed from within local menus.

The UserProc Hook

One other special hook provided by the CommandProcessor is worth noting here, the SetUserHookProc method. This hook allows you to designate a routine that is to be called each time a user's keystroke is evaluated by the GetCommand method. Although there are many

conceivable uses for this hook, probably the most common use for it is to report partial matches, cases where the user has entered the first key of a two-key sequence. For example:

```
{$F+}
procedure MyUserHook(CPP : CommandProcessorPtr; MT : MatchType; Key : Word);
var
  S : string[2];
begin
  S := '  ';
  if MT = PartMatch then
    if Lo(Key) < Ord(' ') then begin
      S[1] := '^';
      S[2] := Char(Lo(Key)+$40);
    end
    else
      S[1] := '+';
  FastWrite(S, 1, 1, $07);
end;
{$F-}
...
Commands.SetUserHookProc(MyUserHook);
```

Such a routine is often useful in cases where many commands are assigned to WordStar-style key sequences such as <CtrlK><CtrlD> and <CtrlQ><CtrlF>. If the user enters <CtrlK>, for example, then '^K' will appear at the top of the screen to remind the user that a partial command has been entered.

See EDITOR.PAS for another example of how to use this hook.

Constants

```
BadCmdOptions : Word = cpDoingChars+cpDeallocP+cpDeallocS;
```

Option settings that exist for internal use only, and may not be altered by calling cpOptionsOn or cpOptionsOff.

```
cpMapWordStar     = $0001; {WordStar mapping on?}
cpEnableMouse     = $0002; {Mouse support enabled?}
cpUpcase          = $0004; {Convert keys to uppercase before evaluating?}
cpWordCmds        = $0008; {If set, commands are words rather than bytes}
cpSwitchPriority = $0010; {Give secondary commands priority when searching?}
cpCtrlsAreChars   = $0020; {Treat unassigned control chars as regular chars}
cpDoingChars      = $2000; {For internal use}
cpDeallocP        = $4000; {For internal use}
cpDeallocS        = $8000; {For internal use}
```

CommandProcessor options that can be enabled/disabled by calling cpOptionsOn or cpOptionsOff. cpMapWordStar indicates that the CommandProcessor should treat key sequences like <CtrlK><D> as being equivalent to <CtrlK><CtrlD>. cpEnableMouse is used to enable or disable mouse support. cpUpcase indicates that alpha keys entered by the user should be uppercased before the command table is scanned; this option should usually be selected if any commands are assigned to alpha keys, such as <A>. cpWordCmds indicates that all the commands in the command table(s) are word-sized rather than byte-sized. cpSwitchPriority indicates that the secondary command table should be scanned before the primary command table. cpCtrlsAreChars indicates that any control characters that are not assigned to specific commands should be treated as ordinary characters. For example, if the user presses <CtrlC>, and <CtrlC> has not been assigned to a command, GetCommand will return ccChar if the option is on, rather than ccNone. The other options are for internal use only.

```
DefCmdOptions : Word = cpMapWordStar;
```

Default options for a CommandProcessor.

Types

```
CommandProcessorPtr = ^CommandProcessor;
CommandProcessor =
  object(Root)
    ...
  end;
```

An object used to process commands entered by the user with the keyboard or a mouse.

```
GetHelpProc = procedure (UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
```

A procedure to be called when the ccHelp command is issued. See the discussion of the "Context Sensitive Help Hook," above, for details.

```
GetKeyProc = function : Word;
```

A function to be called to retrieve the next key pressed by the user. See the discussion of the "Background Task Hook," above, for details.

```
CmdTable = array[0..32000] of Byte;
CmdTablePtr = ^CmdTable;
```

Abstract definition of a command table (a zero-based array of byte). The value 32,000 is arbitrary. A CmdTable is actually of a variable size.

```
KeyPressedProc = function : Boolean;
```

A function to be called to determine whether or not a key has been pressed by the user. See the discussion of the "Background Task Hook," above, for details.

```
MatchType = (NoMatch, PartMatch, FullMatch);
```

Used when determining if a given key code matches any of the commands in a command table. NoMatch means that the key definitely does not correspond to any of the commands. PartMatch indicates a partial match–for example, if <CtrlK> is pressed, and one of the

commands is assigned to <CtrlK><CtrlQ>. FullMatch means that there is a precise match between the key(s) pressed and one of the commands.

```
UserHookProc = procedure (CPP : CommandProcessorPtr; MT : MatchType;
                          Key : Word);
```

A procedure to be called immediately after a keypress has been evaluated. CPP is a pointer to the current command processor. MT indicates whether the key pressed corresponds to a command. Key is the key that was pressed by the user.

Declaration

```
procedure AddCommand(Cmd : Word; NumKeys : Byte; Key1, Key2 : Word);
```

Purpose

Add a new primary command key assignment or change an existing one.

Description

This method allows you to make three kinds of changes to the command key assignments for a CommandProcessor: you can define new commands (called "user exit commands"), disable a command key assignment, or change a command key assignment.

Let's say that you are using OPEDIT and want to allow <F2> to signal that the user would like to see a list of choices. Here's what you would do to get OPEDIT to return control to you when <F2> is pressed:

```
{assign ccUser0 to <F2>}
EditCommands.AddCommand(ccUser0, 1, $3C00, 0);
if EditCommands.GetLastError <> 0 then
  WriteLn('Failed to add command');
```

The first parameter is the command code; here we're using the first of the 55 codes reserved for user exit commands. The second parameter indicates whether the command is being assigned to a single key (such as <F2>) or a 2-key sequence (such as <CtrlQ><D>). The third and fourth parameters give the key codes for the keys. In our example, the command is being assigned to a single key, so the Key2 parameter will be ignored. But if we were adding a 2-key WordStar-type command, say, <CtrlQ><D>, we would pass 'Ord(^Q)' (Key1) and 'Ord(^D)' (Key2). Notice that we would pass 'Ord(^D)' rather than 'Ord(D)': OPEDIT will then accept either <CtrlD> or <D> following the <CtrlQ>, just as WordStar and all the Borland editors would. (This assumes that the cpMapWordStar option is in effect, as it is by default.)

The call to GetLastError is used to determine whether or not the command was added successfully. For a list of possible errors, see the entry for GetLastError.

Now let's suppose that we later wanted to disable our user exit command. To do this we would reassign the <F2> key to the null command, ccNone:

```
{assign <F2> to the null command}
EditCommands.AddCommand(ccNone, 1, $3C00, 0);
```

Besides disabling the exit command, this would also make more room available in the command table for adding other user-defined commands.

Finally, let's say that you want to change an existing key assignment. For example, maybe you want <CtrlHome> to do the same thing as <Home>, and <CtrlEnd> to do the same thing as <End>:

```
{assign <CtrlHome> to ccHome}
EditCommands.AddCommand(ccHome, 1, $7700, $0000);
{assign <CtrlEnd> to ccEnd}
EditCommands.AddCommand(ccEnd, 1, $7500, $0000);
```

When AddCommand scans the command table, it will see that <CtrlHome> is already assigned to ccDelBol and simply change its assignment to ccHome. Similarly, <CtrlEnd> will be assigned to ccEnd rather than ccDelEol. In both cases, the amount of space remaining in the array for expansion will be unchanged.

See Also

AddSecondaryCommand GetLastError

AddSecondaryCommand / CheckForConflict

Declaration

```
procedure AddSecondaryCommand(Cmd : Word; NumKeys : Byte; Key1, Key2 : Word);
```

Purpose

Add a new secondary command key assignment or change an existing one.

Description

This method is identical to AddCommand, except that it modifies the secondary command table rather than the primary command table.

Examples

```
Commands.AddSecondaryCommand(ccUser0, 1, $4400, 0);
if Commands.GetLastError <> 0 then
  WriteLn('Failed to add command');
```

Assign the ccUser0 command to <F10>, previously not associated with a command.

```
Commands.AddSecondaryCommand(ccNone, 2, Ord(^Q), Ord(^I));
```

Disable the command associated with <CtrlQ><CtrlI>.

```
Commands.AddSecondaryCommand(ccUser1, 1, $4700, 0);
```

Change the command associated with the <Home> key to ccUser1.

See Also

AddCommand

Declaration

```
function CheckForConflict(NumKeys : Byte; Key1, Key2 : Word) : MatchType;
```

Purpose

Check to see if the specified key combination conflicts with an existing one.

Description

This method can be used to determine whether or not a given key sequence conflicts with an existing one in either the primary or secondary command tables. The result will be NoMatch if the key sequence has not been assigned to a command, FullMatch if the key sequence has already been assigned, or PartMatch if part of the key sequence has been assigned (for example, if you're checking for <CtrlK><CtrlQ> and a command has been assigned to <CtrlK>).

The three parameters here are the same as the last three parameters to AddCommand: NumKeys is the number of keys in the sequence, Key1 is the first key in the sequence, and Key2 is the second (if NumKeys = 2).

Example

```
if Commands.CheckForConflict(1, $3B00, 0) <> NoMatch then
  WriteLn('<F1> key already assigned')
else
  Commands.AddCommand(ccUser0, 1, $3B00, 0);
```

Assigns the ccUser0 command to the <F1> if the <F1> key is not already associated with a command.

See Also

AddCommand

Declaration

```
function CommandStringPending : Boolean;
```

Purpose

Returns True if a text string is pending in the list of auto-executed commands.

Description

This function is intended to be used by objects such as text editors that might have a text string embedded within a list of commands to be executed automatically. See the section on "Auto-Execution of Commands," above, for details on how this method may be used.

Example

```
const
  CmdList : array[1..8] of Byte = (
    ccDown, ccUp, ccUser0, ccChar, Byte('A'), Byte('B'), Byte('C'), ccChar);
var
  Commands : CommandProcessor;
  Cmd, Key : Word;
begin
  {... initialize Commands ...}
  {point to command list}
  Commands.SetCommandList(@CmdList, 8);
  repeat
    {Get a command or command string}
    if Commands.CommandStringPending then
      Writeln(Commands.GetCommandString)
    else begin
      Cmd := Commands.GetCommand(Key);
      case Cmd of
        ccUp    : WriteLn('ccUp');
        ccDown  : WriteLn('ccDown');
        ccQuit  : WriteLn('ccQuit');
        ccUser0 : WriteLn('ccUser0');
        else      WriteLn('ccNone');
      end;
    end;
  until Cmd = ccQuit;
  Commands.Done;
end;
```

Checks pending commands for commands or embedded strings. Commands are processed; embedded strings are displayed.

See Also

GetCommandString SetCommandList

cpGetHelp / cpGetKey / cpKeyPressed

Declaration
```
procedure cpGetHelp(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word); virtual;
```

Purpose
Called when help is requested.

Description
This method is called by other objects when the user has issued the help command (ccHelp). If a GetHelpProc has been specified by calling SetHelpProc, that procedure will be called with the same parameters passed to cpGetHelp. If no GetHelpProc has been specified, cpGetHelp does nothing.

See Also
SetHelpProc

Declaration
```
function cpGetKey : Word; virtual;
```

Purpose
Called to get next keystroke.

Description
This virtual method is called by GetCommand to retrieve the next keystroke from the user by calling the corresponding GetKeyProc.

cpGetKey may also be called directly by other objects in cases where a key is desired, rather than a command.

Example
```
WriteLn('Press any key to continue...');
Key := Commands.cpGetKey;
```

See Also
SetGetKeyProc

Declaration
```
function cpKeyPressed : Boolean; virtual;
```

Purpose
Called to determine if a key has been pressed.

Description
This virtual method is never called directly by a CommandProcessor, only by certain objects and routines that use a CommandProcessor. It returns True if the corresponding KeyPressedProc reports that a key is waiting to be processed.

Example
```
while not Commands.cpKeyPressed do
  {do something while waiting for a key to be pressed} ;
Cmd := Commands.GetCommand(Key);
```

Perform some task while waiting for a key to be pressed. For an example of an actual case where cpKeyPressed is needed, see OPMACED.PAS.

See Also
SetKeyPressedProc

Declaration
```
function cpOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if the specified option(s) are selected.

Example
```
if Commands.cpOptionsAreOn(cpSwitchPriority) then
  Write('Secondary')
else
  Write('Primary');
Writeln (' command table will be searched first.');
```
Displays the state of the cpSwitchPriority option.

Declaration
```
procedure cpOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadCmdOptions (see the Declarations section, above). Note that, if cpEnableMouse is one of the options in OptionFlags, the GetKeyProc for the CommandProcessor will be reset to point to OpCrt.ReadKeyWord and the KeyPressedProc will point to OpCrt.KeyPressed.

Example
```
Commands.cpOptionsOff(cpSwitchPriority+cpEnableMouse);
```
Turns off option to search secondary command first before primary. Also turns off mouse support.

Declaration
```
procedure cpOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadCmdOptions (see the Declarations section, above). Note that, if cpEnableMouse is one of the options in OptionFlags, the GetKeyProc for the CommandProcessor will be reset to point to OpMouse.ReadKeyOrButton and the KeyPressedProc will point to OpMouse.KeyOrButtonPressed.

Example
```
Commands.cpOptionsOn(cpSwitchPriority+cpEnableMouse);
```
Turns on option to search secondary command table before the primary command table. Also turns on mouse support.

cpUserHook / Done

Declaration
```
procedure cpUserHook(MT : MatchType; Key : Word); virtual;
```

Purpose
Called each time a keystroke is processed.

Description
This method, intended primarily for internal use, is called each time a keystroke is processed by GetCommand. If a UserHookProc has been specified by calling SetUserHookProc, that routine is called with the same parameters passed to cpUserHook. If not, cpUserHook does nothing.

See Also
SetUserHookProc

Declaration
```
destructor Done; virtual;
```

Purpose
Dispose of any memory that was allocated by the CommandProcessor.

Description
This destructor deallocates the memory for the command table(s) if appropriate. It needs to be called only if the CommandProcessor was loaded from a stream.

See Also
Store

Declaration

```
function GetCmdAssignment(NumKeys : Byte; Key1, Key2 : Word;
                          var Primary : Boolean) : Word;
```

Purpose

Return command associated with specified key combination.

Description

This method may be used to determine what command, if any, a particular key sequence is associated with. If no command has been assigned to it, the function result will be 0 (ccNone).

The first three parameters here are the same as the last three parameters to AddCommand: NumKeys is the number of keys in the sequence, Key1 is the first key in the sequence, and Key2 is the second (if NumKeys = 2).

If the result is something other than ccNone, then Primary indicates whether the command assignment was found in the primary command table (Primary = True) or the secondary command table (Primary = False).

Example

```
var
  Cmd : Word;
  Primary : Boolean;
...
  {What command is associated with <F1>?}
  Cmd :=  Commands.GetCmdAssignment(1, $3B00, 0, Primary);
  if Cmd = ccNone then
    WriteLn('<F1> not used')
  else begin
    Write('<F1> returns ', Cmd, ' from ');
    if Primary then
      WriteLn('primary command processor')
    else
      WriteLn('secondary command processor');
  end;
```

This example displays the command associated with the <F1> key. It also reports where the command was found (primary or secondary command table).

See Also

AddCommand GetKeyAssignment

GetCommand / GetCommandString

Declaration

```
function GetCommand(var Key : Word) : Word;
```

Purpose

Get next command.

Description

This method is called to obtain the next command from the user. If the key(s) pressed correspond to a command, that command is returned as the function result. If not, the result will be either ccChar--meaning that the key pressed was a regular character (' '..'~', #128..#255)--or ccNone (0). (ccChar will also be returned for #0..#31 if the cpCtrlsAreChars option is on.)

The last key pressed by the user is returned in the Key parameter. It is usually of interest only if GetCommand returns ccChar.

Normally GetCommand searches the primary command table first, then the secondary command table, if any. If the cpSwitchPriority option has been set, however, the secondary command table will be searched first. The order is relevant only in cases where the same key sequence has been assigned to different commands in the two command tables.

Other options that affect GetCommand: If the cpUpcase option is in effect, GetCommand will convert the Key pressed by the user to uppercase before scanning the command table(s). If the cpWordCmds option is in effect, GetCommand will assume that commands in both the primary and secondary command tables are represented as words (two bytes) rather than bytes. If the cpMapWordStar option is in effect, alphabetic characters will be converted to their control character equivalents if the previous key produced only a partial match with any of the commands in the command table(s).

After each keypress has been evaluated, GetCommand calls the cpUserHook method with the results of that evaluation, giving the UserHookProc an opportunity to report partial matches.

Example

```
repeat
  Cmd := Commands.GetCommand(Key);
  case Cmd of
    {...process the command...}
  end;
until (Cmd = ccQuit) or (Cmd = ccError);
```

Process commands until <Esc> is pressed or an error occurs.

See Also

cpUserHook

Declaration

```
function GetCommandString : string;
```

Purpose

Returns a text string when CommandStringPending is True.

Description

This method should be called when CommandStringPending is True; it returns the string embedded within the sequence of commands. See the discussion of "Auto-Execution of Commands" for details.

Example

See the example for CommandStringPending.

See Also

CommandStringPending SetCommandList

Declaration

```
procedure GetKeyAssignment(Cmd : Word; var NumKeys : Byte;
                           var Key1, Key2 : Word);
```

Purpose

Return one set of key(s) assigned to the specified command.

Description

This method scans the primary and (if necessary) secondary command tables to find the key sequence associated with the specified command (Cmd). If no match is found, NumKeys will be set to 0. Otherwise, NumKeys returns the number of keys in the sequence (1 or 2), and Key1 and Key2 contain the keys themselves.

Note that the primary command table is always searched first, even if the cpSwitchPriority option is in effect, and that only the first key sequence found is reported (that is, other key sequences may also be associated with the same command).

Example

```
var
  NumKeys : Byte;
  Key1, Key2 : Word;
...
  Commands.GetKeyAssignment(ccRight, NumKeys, Key1, Key2);
  if NumKeys <> 0 then begin
    WriteLn('ccRight is an active command');
```

Searches the primary and secondary command tables for the ccRight command.

See Also

GetCmdAssignment

Declaration

```
function GetKeyPtr(var MaxIndex : Word) : CmdTablePtr;
```

Purpose

Return a pointer to the primary command table.

Description

This function returns a pointer to the primary command table. MaxIndex is the index for the last element in the command table. That is, if the command table is an 'array[1..n] of Byte', then MaxIndex will return n.

Example

```
var
  SavePtr : Pointer;
  SaveMax : Word;
...
  SavePtr := Commands.GetKeyPtr(SaveMax);
  ...
  Commands.SetKeyPtr(SavePtr, SaveMax);
```

Saves a pointer and maximum index to the existing command table and later restores it. Presumably, an alternate command table is used in the interim.

See Also

GetSecondaryKeyPtr SetKeyPtr

GetLastError / GetSecondaryKeyPtr

Declaration

```
function GetLastError : Word;
```

Purpose

Returns last error code and clears it.

Description

This method should be called after a call to AddCommand or AddSecondaryCommand to determine whether or not a command was added successfully to the command table. Possible values are:

```
0                          Success
epNonFatal+ecKeyTableFull  Table is full
epNonFatal+ecPartialMatch  Key sequence partially matches an existing one
epNonFatal+ecBadParam      The NumKeys parameter was not 1 or 2
```

The second error, ecPartialMatch, would occur if you tried to assign a command to <CtrlK>, for example, when another command was assigned to <CtrlK><CtrlQ>.

Example

```
Commands.AddCommand(ccUser0, 1, $3B00, 0);
if Commands.GetLastError <> 0 then
  {unable to assign ccUser0 command to <F1>} ;
```

See Also

AddCommand

Declaration

```
function GetSecondaryKeyPtr(var MaxIndex : Word) : CmdTablePtr;
```

Purpose

Return a pointer to the secondary command table.

Description

This function returns a pointer to the secondary command table if there is one, or nil if there isn't. MaxIndex is the index for the last element in the command table. That is, if the command table is an 'array[1..n] of Byte', then MaxIndex will return n.

Example

```
var
  SavePtr : Pointer;
  SaveMax : Word;
...
  SavePtr := Commands.GetSecondaryKeyPtr(SaveMax);
  ...
  Commands.SetSecondaryKeyPtr(SavePtr, SaveMax);
```

Saves a pointer and maximum index to the existing secondary command table and later restores it. Presumably, an alternate command table is used in the interim.

See Also

GetKeyPtr SetSecondaryKeyPtr

Declaration

```
constructor Init(KeysPtr : CmdTablePtr; MaxIndex : Word);
```

Purpose

Initialize a command processor with only one command table.

Description

This constructor initializes a "regular" CommandProcessor, one that has only a single (primary) command table. KeysPtr points to the start of the command table, and MaxIndex specifies the last index into the array.

Note that it is not necessary to treat Init as a boolean function, since it will never fail.

Example

See the example for InitCustom, below.

Declaration

```
constructor InitCustom(PrimaryKeys : CmdTablePtr; PrimaryMaxIndex : Word;
                       SecondaryKeys : CmdTablePtr; SecondaryMaxIndex : Word;
                       Options : Word);
```

Purpose

Initialize a command processor with custom options.

Description

This constructor initializes a "special" CommandProcessor, one that has two command tables (primary and secondary) or one that requires special options.

PrimaryKeys points to the start of the primary command table, and PrimaryMaxIndex specifies the last index into the array. SecondaryKeys points to the start of the secondary command table, and SecondaryMaxIndex specifies the last index into it. If there is no secondary command table, pass 'nil, 0'. Options are the CommandProcessor options that are to be in effect; if no special options are required, use 'DefCmdOptions'.

Note that it is not necessary to treat InitCustom as a boolean function, since it will never fail.

Example

```
const
  {Define a small KeySet of a few cursor commands}
  KeyMax  = 30;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys          command type   key sequence}
  3,      $00, $48,     ccUp,          {Up}
  3,      $00, $50,     ccDown,        {Down}
  3,      $00, $4B,     ccLeft,        {Left}
  3,      $00, $4D,     ccRight,       {Right}
  2,      $1B,          ccQuit,        {Esc}
  0,0,0,0,0,0,0,0,0,0,0,0);            {<-room for more commands}
  SecKeyMax = 15;
  SecKeySet : array[0..SecKeyMax] of Byte = (
  3,      $00, $47,     ccHome,        {Home}
  3,      $00, $4F,     ccEnd,         {End}
  2,      $1B,          ccQuit,        {Esc}
  0,0,0,0,0);                          {<-room for more commands}
var Commands : CommandProcessor;
...
Commands.InitCustom(@KeySet, KeyMax,           {Primary keys}
                    @SecKeySet, SecKeyMax,     {Secondary keys}
                    DefCmdOptions);            {Options}
```

Create a CommandProcessor with both primary and secondary command tables.

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a command processor from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to CommandProcessor.Store. If the command table(s) for the CommandProcessor were not registered at the time they were stored, and the contents of the command tables was consequently stored in the stream, Load will allocate space for the command table(s) and load them into memory.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a CommandProcessor is CommandProcessorStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd;
var
  Commands : CommandProcessor;
  Cmd, Key, Status : Word;
  S : BufIdStream;
begin
  S.Init('COMMAND.STM', SOpen, 1024);
  S.RegisterHier(CommandProcessorStream);
  S.Get(Commands);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Load error ', Status);
    Halt;
  end;
  S.Done;
  repeat
    Cmd := Commands.GetCommand(Key);
    case Cmd of
      ccUp    : WriteLn('ccUp');
      ccDown  : WriteLn('ccDown');
      ccLeft  : WriteLn('ccLeft');
      ccRight : WriteLn('ccRight');
      ccQuit  : WriteLn('ccQuit');
      ccUser0 : WriteLn('ccUser0');
      else      WriteLn('ccNone');
    end;
  until Cmd = ccQuit;
  Commands.Done;
end.
```

Instantiates Commands from the stream created in the Store example (Load is called indirectly by the stream's Get). Note that, in that example, the command table was not registered, so no command table is needed here.

See Also

Store

Declaration
```
function MouseEnabled : Boolean;
```

Purpose
Return True if mouse support is enabled.

Description
This function returns True if mouse support is enabled for the CommandProcessor. Its result is identical to that of the expression "cpOptionsAreOn(cpEnableMouse)".

See Also
cpOptionsAreOn

Declaration
```
procedure SetCommandList(CmdList : Pointer; NumCmds : Word);
```

Purpose
Specify a list of commands to be auto-executed.

Description
This method allows you to specify a list of commands that are to be executed automatically. CmdList is a pointer to that list, which should be either an array of bytes or, if the cpWordCmds option is selected, an array of words. NumCmds is the number of commands in the list. See the discussion of "Auto-Execution of Commands" at the beginning of this section for details.

Example
See the example for CommandStringPending.

See Also
CommandStringPending GetCommandString

SetGetKeyProc / SetHelpProc

Declaration

```
procedure SetGetKeyProc(GKP : GetKeyProc);
```

Purpose

Specify routine to call to get next key.

Description

This routine allows you to specify the routine to be called by cpGetKey to get the next keystroke from the user. A typical use for it is in applications where a background task needs to be performed while waiting for keystrokes. See the discussion of the "Background Task Hook" at the beginning of this section for details.

Note that, if you are enabling mouse support with the "cpMouseSupport" option, this routine must be called after cpOptionsOn.

Example

```
uses OpCrt, OpCmd, OpDate;
{$F+}
function OurGetKey : Word;
begin
  while not KeyPressed do begin
    inline($CD/$28);      {Give TSRs a chance to pop up}
    FastWrite(CurrentTimeString('HH:MM'), 1, 75, TextAttr); {Update screen
clock}
    {...perform other background tasks...}
  end;
  OurGetKey := ReadKeyWord;
end;
{$F-}
...
Commands.SetGetKeyProc(OurGetKey);
```

This is a replacement function for GetKey that updates an on-screen clock while waiting for keystrokes.

See Also

cpGetKey SetKeyPressedProc

Declaration

```
procedure SetHelpProc(GHP : GetHelpProc);
```

Purpose

Specify routine to call to request help.

Description

This method allows you to specify the routine to be called when the user requests help. See the discussion of the "Context Sensitive Help Hook" at the beginning of this section.

Example

```
{$F+}
procedure MyHelpProc(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
begin
  {Show help based on UnitCode, IdPtr and HelpIndex}
end;
{$F-}
...
Commands.SetHelpProc(MyHelpProc);
```

See Also

cpGetHelp

Declaration
```
procedure SetKeyPressedProc(KPP : KeyPressedProc);
```

Purpose
Specify routine to call to determine if key has been pressed.

Description
This method lets you specify the routine that is to be called by cpKeyPressed (normally OpCrt.KeyPressed or OpMouse.KeyOrButtonPressed). The hook it provides is normally unnecessary. One case where it would be needed, however, would be in program that uses OPMACED and needs to perform a background task while waiting for a keystroke.

Example
```
{$F+}
function MyKeyPressedProc : Boolean;
begin
  MyKeyPressedProc := KeyPressed;
end;
{$F-}
...
Commands.SetKeyPressedProc(MyKeyPressedProc);
```

See Also
cpKeyPressed SetGetKeyProc

Declaration
```
procedure SetKeyPtr(KeysPtr : CmdTablePtr; MaxIndex : Word);
```

Purpose
Change the primary command table.

Description
This method allows you to change the primary command table for a CommandProcessor. The most common use for it is in situations where one command table is to be used for one instance of an object type, and another for a second. For example, you might want to use a special command table for a particular DirList (OPDIR), rather than the ones normally used for an object descended from a PickList (OPPICK).

KeysPtr is the address of the command table, and MaxIndex is the last index in the array.

Example
```
Commands.SetKeyPtr(KeySet3, KeySetMax3);
```

This assumes the existence of a command table named KeySet3, an 'array[0..KeySetMax3] of Byte'. After this call, KeySet3 will be the primary command table.

See Also
SetSecondaryKeyPtr

SetSecondaryKeyPtr / SetUserHookProc

Declaration

```
procedure SetSecondaryKeyPtr(KeysPtr : CmdTablePtr; MaxIndex : Word);
```

Purpose

Change the secondary command table.

Description

This method allows you to specify a secondary command table. The most common use for it is in applications that have multiple window-based objects active at once, and a common set of "hot keys" is needed for all of them. DESKPOP is an example of such an application. See TPUI.PAS for a demonstration of how to use SetSecondaryKeyPtr.

KeysPtr is the address of the command table, and MaxIndex is the last index in the array.

Example

```
Commands.SetSecondaryKeyPtr(KeySet3, KeySetMax3);
```

This assumes the existence of a command table named KeySet3, an 'array[0..KeySetMax3] of Byte'. After this call, KeySet3 will be the secondary command table.

See Also

SetKeyPtr

Declaration

```
procedure SetUserHookProc(UHP : UserHookProc);
```

Purpose

Specify a routine to be called each time a keystroke is evaluated.

Description

This method allows you to specify a routine that is to be called each time a keystroke is evaluated by GetCommand.

The most common use for such a routine is to inform the user that a partial command has been entered. For example, if there are several two-key commands that start with <CtrlK>, the routine might want to display '^K' on a status line whenever that key is pressed. See EDITOR.PAS for an example of such a routine.

Examples

```
{$F+}
procedure MyUserHook(CPP : CommandProcessorPtr; MT : MatchType; Key : Word);
begin
  if MT = PartialMatch then
    RingBell;
end;
{$F-}
...
Commands.SetUserHookProc(MyUserHook);
```

Shows how to install a procedure that gets called whenever a command is processed. This example beeps whenever a partial command has been entered by the user.

```
Commands.SetUserHookProc(@MyUserHook);
```

Incorrect! The parameter to SetUserHook is a procedure type, not the address of a procedure.

See Also

cpUserHook GetCommand

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a command processor in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the CommandProcessor to the stream S. If the command table(s) associated with it have not been registered, their contents will also be stored in the stream along with the object.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a CommandProcessor is CommandProcessorStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd;
const
  {Define a small KeySet of a few cursor commands}
  KeyMax  = 30;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys         command type   key sequence}
  3,      $00, $48,    ccUp,          {Up}
  3,      $00, $50,    ccDown,        {Down}
  3,      $00, $4B,    ccLeft,        {Left}
  3,      $00, $4D,    ccRight,       {Right}
  2,      $1B,         ccQuit,        {Esc}
  0,0,0,0,0,0,0,0,0,0,0,0);           {<-room for more commands}
var
  Commands : CommandProcessor;
  Cmd, Key, Status : Word;
  S : BufIdStream;
begin
  Commands.Init(@KeySet, KeyMax);
  S.Init('COMMAND.STM', SCreate, 1024);
  S.RegisterHier(CommandProcessorStream);
  S.Put(Commands);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  Commands.Done;
end.
```

Instantiates Commands and stores it to a stream using the stream's Put method, which calls CommandProcessor.Store indirectly. See Load for an example of how to reload this stream.

See Also

Load

3. The CommandPacker

Earlier in the chapter we mentioned that one of the benefits of using command tables is that it simplifies the task of making key assignments customizable. That's true, but there's nothing in a CommandProcessor that helps you write installation programs. For that task, you have to use a second object, called a CommandPacker, which provides methods for "unpacking" a command table so that the key assignments can be easily modified, then repacking them after they have been modified. A simple example is shown below.

```
program CommandPackerDemo; {EXCMDPK.PAS}
uses OpCrt, OpCmd;
const
  {Define KeySet of a few cursor commands}
  KeyMax  = 30;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys         command type   key sequence}
  2,      $1B,         ccQuit,        {Esc}
  3,      $00, $4B,    ccLeft,        {Left}
  3,      $00, $4D,    ccRight,       {Right}
  3,      $00, $48,    ccUp,          {Up}
  3,      $00, $50,    ccDown,        {Down}
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);   {<-room for more commands}

  MaxCmd  = ccDown;                   {highest legal command in command table}
  NumCols = 3;                        {number of columns in unpacked command table}
var
  Packer : CommandPacker;             {CommandPacker object}
  UKeySet : array[1..MaxCmd, 1..NumCols] of KeyRec; {Unpacked command table}
  Commands : CommandProcessor;        {CommandProcess object}
  Key, Cmd : Word;                    {last key, command}
begin
  {Unpack the KeySet command table}
  Packer.Init(@KeySet,    {Pointer to packed command table}
              MaxCmd,     {Highest legal command in packed command table}
              KeyMax,     {Highest legal index into packed command table}
              @UKeySet,   {Pointer to unpacked command table}
              NumCols,    {Number of "columns" in unpacked command table}
              False);     {Commands are bytes, not words}

  {UKeySet is now available for modification...change ccUp from <Up> to <F1>}
  UKeySet[ccUp, 1].Keys := #$00#$3B;
  UKeySet[ccUp, 1].Modified := True;

  {Check the modified UKeySet for keystroke conflicts}
  if Packer.ConflictsFound then
    WriteLn('Keystroke conflict was found');

  {Repack the UKeySet array into the KeySet array}
  if not Packer.PackKeys then
    Writeln('Failed to repack command table')
  else
    Writeln('Keys repacked successfully');

  {Demonstrate our custom command table}
  Commands.Init(@KeySet, KeyMax);
  repeat
    Cmd := Commands.GetCommand(Key);
    case Cmd of
      ccUp    : WriteLn('ccUp');
      ccDown  : WriteLn('ccDown');
```

```
        ccLeft  : WriteLn('ccLeft');
        ccRight : WriteLn('ccRight');
        ccQuit  : WriteLn('ccQuit');
        else      WriteLn('ccNone');
      end;
    until Cmd = ccQuit;
  end.
```

The first thing to notice here is the two-dimensional UKeySet array, which is used to hold the key assignments after the command table (KeySet) has been unpacked. After unpacking, the key assignments are arranged in a series of three "columns." This means that, if ccQuit were assigned to three different key sequences, then after unpacking the three key sequences would be stored in UKeySet[ccQuit,1], UKeySet[ccQuit,2], and UKeySet[ccQuit,3]. The first dimension of the array indicates the highest command code that needs to be stored in it. Typically the highest command will be a user exit command, such as ccUser0 (200), but in this simple case the highest is ccDown (13).

Each element in the unpacked command table (UKeySet) is a record of type KeyRec:

```
const
  KeyLength = 6;
type
  KeyString = string[KeyLength];
  KeyRec =
    record
      Modified : Boolean;
      Conflict : Boolean;
      CmdCode : Word;
      Keys : KeyString;
    end;
```

The first field, Modified, is unused by the CommandPacker; it just sets aside space for flagging commands that have been modified by the user. The second field, Conflict, is initialized to False by Init, but can be set to True by the ConflictsFound method, which checks for conflicts between key assignments. The third field, CmdCode, stores the command code that goes with the key sequence, which is stored as a string of up to six characters in the fourth field, Keys.

After the command table has been unpacked by Init, the key assignment for ccUp is changed from <Up> to <F1>. This is obviously an overly simplistic example--normally a configuration routine would allow key assignments to be changed interactively by the user--but it does demonstrate what needs to be changed.

The next step is to check for conflicts. Here too the process is over-simplified. A configuration routine would typically refuse to continue after conflicts are found, and would offer the user an opportunity to resolve the conflicts.

The final step is to repack the keys into the original command table using the PackKeys method. If the operation is unsuccessful, it returns False, indicating that the command table is too small to hold all the key assignments. Although the example program continues at this point, the proper response is to display an error message and allow the user to eliminate some of his key assignments.

Obviously, our little example program leaves much to be desired. The point of the example is to show you the basic steps involved in modifying a command table--what methods you need to call, and the order in which they should be called. For a more elaborate demonstration of how to use a CommandPacker, see DPINST.PAS.

Constants

```
KeyLength = 6;
```

The maximum length of a KeyString.

```
MaxCommands = 255*3;
```

Stated maximum number of commands in an unpacked command table. Value is arbitrary; the array can actually be larger if necessary.

Types

```
CommandPacker =
  object(Root)
    ...
  end;
```

An object used to unpack and repack command tables used by CommandProcessors.

```
KeyString = string[KeyLength];
```

A string representation of a key sequence.

```
KeyRecPtr = ^KeyRec;
KeyRec =
  record
    Modified : Boolean;
    Conflict : Boolean;
    CmdCode : Word;
    Keys : KeyString;
  end;
```

An element in an unpacked command table. CmdCode is the code for the command whose key assignment is to be modified. Keys is a string representing the keys assigned to it: for example, if the command is assigned to <F10>, whose scan code is $4400, Keys would contain '#0#$44' (two characters); if the command is assigned to <CtrlK><CtrlD>, Keys would contain '^K^D' (two characters). If no assignment has been made, the string would be empty. The Conflict field is set to False when the KeyRec is initialized, and is set to True only if ConflictsFound determines that the key assignment creates a conflict. The Modified field is set to False when the KeyRec is initialized, and its value is never changed by any of the methods in the CommandPacker. It simply sets aside space for user-written routines that need to flag key assignments that have been changed.

```
UnpackedCmdTable = array[1..MaxCommands] of KeyRec;
UnpackedCmdPtr = ^UnpackedCmdTable;
```

A command table after unpacking.

Declaration

```
function ConflictsFound : Boolean;
```

Purpose

Check unpacked command table for conflicts.

Description

This method is used to determine whether or not there are any conflicts in the unpacked command table. For example, if the user has requested that <F1> be assigned to both ccHelp and ccQuit, that would create a conflict. ConflictsFound should therefore be called before PackKeys, and if it returns True the user should be forced to resolve the conflict(s).

After ConflictsFound has been called, each KeyRec involved in a conflict will have its "Conflict" flag set to True.

Example

```
UKeySet : array[1..MaxCmd, 1..NumCols] of KeyRec; {Unpacked command table}
Cmd, Col : Byte;
...
if Packer.ConflictsFound then begin
  WriteLn('Keystroke conflicts found for the following command codes:');
  for Cmd := 1 to MaxCommands do
    for Col := 1 to NumCols do
      if UKeySet[Cmd, Col].Conflict then
        WriteLn('Command: ', Cmd, '  Column: ', Col);
end;
```

Checks the unpacked command table for conflicts and lists all commands

---

Declaration

```
function GetKeyRecPtr(Cmd : Word; Col : Byte) : KeyRecPtr;
```

Purpose

Returns a pointer to the KeyRec for the Col'th instance of Cmd.

Description

This method allows you to locate a given key assignment in an unpacked command table. It is useful primarily in cases where the unpacked command table was declared as a one-dimensional array, and in general purpose routines that do not have direct access to the unpacked command table.

If the unpacked command table is declared as

```
var
  UnpackedKeys : array[1..MaxCmd, 1..NumCols] of KeyRec;
```

and Cmd and Col are both legal indexes into the array, then GetKeyRec will return a pointer to UnpackedKeys[Cmd, Col]. If either Cmd or Col is out of bounds, it will return nil.

Example

```
var
  KRP : KeyRecPtr;
...
KRP := Packer.GetKeyRecPtr(ccUp, 1);
```

Returns a pointer to the first KeyRec for the ccUp command.

Init / PackKeys / SizeKeys

Declaration

```
constructor Init(PKeys : CmdTablePtr; MaxCmd, MaxIndex : Word;
                 UKeys : UnpackedCmdPtr; Columns : Byte; CmdsAreWords : Boolean);
```

Purpose

Initialize pointers to packed/unpacked command tables.

Description

This constructor initializes a CommandPacker and unpacks the keys in the packed command table passed to it. PKeys is the address of the unpacked command table, and MaxIndex is the last legal index into it. MaxCmd is the value of the highest command stored in the array. UKeys is the address of the array in which to store the key assignments after they have been unpacked. Columns is the number of "columns" in the unpacked command table. And CmdsAreWords indicates whether or not the commands in the packed command table are stored as words (True) rather than bytes (False).

More precisely, PKeys is assumed to point to an array of the form

```
PackedKeys : array[0..MaxIndex] of Byte;
```

and UKeys is assumed to point to an array of the form

```
UnpackedKeys : array[1..MaxCmd, 1..Columns] of KeyRec;
```

Note that Init never fails. It assumes that the unpacked command table is as large as it is supposed to be, and if MaxCmd is too low--that is, if the packed command table assigns a key sequence to a command higher than MaxCmd--the error will be ignored.

Example

See the example at the beginning of this section of the manual.

Declaration

```
function PackKeys : Boolean;
```

Purpose

Convert unpacked array into a packed list of keys again.

Description

This method is to be called after the unpacked command table has been modified, and its contents need to be repacked. It returns True if the operation succeeds, or False if the keys won't all fit in the space allocated to the packed table. If the operation fails, you can use SizeKeys to determine how close the keys are to fitting.

Note that PackKeys should normally not be called until after ConflictsFound has been called to determine whether or not any conflicts exist in the unpacked command table.

Example

```
if not Packer.PackKeys then
  Writeln('Failed to repack command table')
else
  Writeln('Keys packed successfully');
```

Declaration

```
function SizeKeys : Word;
```

Purpose

Return number of bytes in packed version of the unpacked command table.

Description

SizeKeys calculates the size of an unpacked command table after packing. A typical use for it is in a case where PackKeys has failed and you need to report how many bytes too many are in the unpacked command table.

4. Procedures and Functions

This section documents miscellaneous procedures and functions found in OPCMD.

DefaultErrorProc / RingBell / SetBell

Declaration

```
procedure DefaultErrorProc(UnitCode : Byte; var ErrorCode : Word;
                           ErrorMsg : string);
```

Purpose

Default error handler for all objects.

Description

This routine is the default error handler for all objects that have one. It simply calls RingBell.

Declaration

```
procedure RingBell;
```

Purpose

Ring the bell.

Description

This routine makes a sound with a default pitch of 700 hertz, and a default duration of 100 milliseconds (1/10 of a second). These values may be changed by calling SetBell.

Example

```
{$F+}
procedure MyErrorProc(UnitCode : Byte;
                      var ErrorCode : Word; ErrorMsg : string);
begin
  if ErrorMsg <> '' then
    WriteLn(ErrorMsg)
  else
    WriteLn('Error ', ErrorCode);
  RingBell;
end;
{$F-}
```

A crude error handler.

Declaration

```
procedure SetBell(Pitch, Duration : Word);
```

Purpose

Set pitch and duration for bell.

Description

This routine sets the pitch and duration of the sound made by RingBell. Pitch indicates the frequency of the sound, in hertz, and Duration the length of time, in milliseconds, for which the sound is to continue.

Example

```
SetBell(500, 150);
RingBell;
```

Lower the pitch of the "bell" and increase its duration, then ring it.

4. Window Objects

A. Overview

The Window originated as a lowly concept in Turbo Pascal 2.0. At that time, a window represented just a pair of X,Y coordinates which specified a screen region where writing and scrolling would occur. Soon, add-on packages were developed to extend this concept with frames around the text area and the ability to save and restore the screen contents beneath the window.

With the advent of separately compiled units in Turbo Pascal 4.0, the window concept was ready for another step forward in sophistication. Turbo Professional 4.0, among other packages, provided windows that could be stacked and erased, shadowed and exploded, moved and resized. Turbo Professional 5.0 added higher level units: for example, pick lists and help text could be displayed within windows, and many of the basic windowing capabilities--frames, shadows, titles, etc.--were available to these units.

Now, object-oriented programming enables the next great leap in window power. Consider a primitive window as an object that knows how to draw, erase, move, and resize itself. This window knows little about its actual contents, and that works to our benefit. When we derive more specialized objects from this primitive window, each derived object inherits the capabilities of its ancestor and then adds its own new ones. In this way, an entire hierarchy of window objects--first primitive, then stackable, then with an interface to the keyboard and mouse, then containing pick lists, help systems, data entry screens, and many more capabilities--can be used in a self-consistent fashion, mixed and matched as required by an application.

User interface design is an ideal application of object-oriented programming methodology. We've taken that statement to heart in **Object Professional** 1.0 and the result is a total reorganization of the windowing capabilities of Turbo Professional. If you're familiar with Turbo Professional, you'll certainly notice the similarities in this new package. You'll also notice the many powerful extensions and, just as important, the unification of many concepts which had previously been "tacked on" as Turbo Professional matured. We've become believers in OOP after seeing the capabilities that are possible with a carefully designed hierarchy such as **Object Professional**'s windows represent.

This chapter describes much of the window object hierarchy, including the following object types:

```
[Root]
  LoadableColorSet
  PackedWindow
  [PointerStack]
    WindowStack
  [DoubleListNode]
    HeaderNode*
    ShadowNode*
    HotNode*
    MenuItemNode
  [DoubleList]
    [CircularList]
      MenuItemList*
  ScrollBar*
  ScreenRect*
  VirtScreen
```

```
AbstractFrame
  Frame
AbstractWindow
  RawWindow
    SubMenu*
    StackWindow
      CommandWindow
        Menu
        PickList
          DirList
          PathList
          AbstractHelpWindow
            PagedHelpWindow
            ScrollingHelpWindow
```

The objects whose names are surrounded by brackets are described elsewhere. In this case all of them are described with the OPROOT unit (Chapter 8). Objects followed by an asterisk are low-level ones primarily designed for internal use. Methods of these objects that are useful to application programmers will be described as appropriate. The window hierarchy even extends beyond what is shown here; the remainder applies to data entry screens (EntryScreen, ScrollingEntryScreen) and other high level specialized windows (Memo, TextEditor, Browser, and others), described elsewhere.

A brief overview of each object here will allow you to choose the appropriate one for your needs. The following sections provide detailed descriptions of each one.

The LoadableColorSet is a simple extension of the ColorSet object described with the OPCRT unit. It contains a complete ColorSet as a data field, which is used to describe the video attributes of any **Object Professional** screen object. Unlike ColorSet, LoadableColorSet is derived from Root and provides Load and Store methods. As such, it can be written to and read from streams. Otherwise, it has the same features as a ColorSet.

PackedWindow is an object-oriented version of the facility distributed with Turbo Professional's TPCRT unit. A PackedWindow stores the contents of a rectangular screen region in a compressed format. Space savings due to compression will typically range from 30% to 90%. Such packed windows may be written to and read from disk, linked directly into an EXE file, and displayed on the screen. If you have a large number of static screens to display, PackedWindows are a good choice to use. We provide a memory-resident utility called PSCREEN, which allows you to capture text screens created by any program and store them to disk in a PackedWindow format. We'll describe how to use PSCREEN in section H of this chapter.

A WindowStack is a specialized form of PointerStack (from OPROOT). A WindowStack manages the relationships between all StackWindows simultaneously visible on the screen. **Object Professional** declares a static global WindowStack variable called wStack. All windows of type StackWindow (or derivatives thereof) are automatically pushed on wStack when they are drawn and popped when erased. Most uses of WindowStack are automated by the StackWindow itself; WindowStack methods include ones to erase and pop the top window on the stack and to evaluate a particular mouse position to determine which window it falls into.

The window objects make substantial use of linked lists to manage collections of elements whose number isn't known in advance. HeaderNode, ShadowNode, HotNode, and MenuItemNode are all derived from the DoubleListNode to serve as nodes in these linked lists. A HeaderNode manages a window header, which includes a position, a string, and video attributes. A ShadowNode manages a rectangular portion of the screen, positioned adjacent to

a window frame and capable of drawing itself as a see-through or opaque shadow. A HotNode specifies another screen rectangle; this one may be used to detect mouse clicks and to map those clicks to command events. A MenuItemNode represents one item in a menu window. It has a position, a string, a selection character, and an optional help string. All of these node types provide methods to draw, modify, and return information about themselves. In most cases you won't refer to the nodes directly, but the methods documented in the following sections will show those cases where it may be important.

The MenuItemList is a specialized linked list to manage the items in one submenu of a menu system. It describes the orientation of the list, its video attributes, the currently selected item, some user-supplied customization routines, as well as the list of MenuItemNodes. You probably won't need to access this object directly since the Menu manages it for you.

A ScrollBar is a primitive object that manages the mapping and display of mouse scrollbars. It describes the visual characteristics of the scrollbar—position, orientation, characters, and attributes—as well as the range of user values that correspond to the full range of the bar. The ScrollBar provides methods to draw and erase the "slider" (which marks the relative position of the "current element" in the window; for example, if the highlight bar in a pick list were on the 50th element of 100 total, the slider would appear halfway down the scrollbar). You'll typically access the ScrollBar through methods of the Frame object.

ScreenRect implements some of the actions originally associated with windows, specifically those to save and restore rectangular regions of the screen while avoiding video snow in the process. Each RawWindow object includes two ScreenRect objects, one to manage the portion of the screen covered by the window when it is visible, and an optional one to save the contents of the window itself. Unless you're defining completely new categories of windows, you probably won't need to access a ScreenRect directly.

VirtScreen plays an important supporting role in many **Object Professional** screen operations. A VirtScreen manages an area of memory as if it were the physical display buffer of a video adapter card. The contents of a virtual screen can be initialized from the physical screen and copied back to it. All of OPCRT's FastWrite family of screen writing routines, and all of the window objects' screen update methods, can be directed to a virtual screen instead of the physical one. **Object Professional** uses virtual screens internally to achieve fantastic visual effects with no unsightly screen flash when windows are moved or resized, a window is pulled to the top of the stack, a hidden window is written to, or a large entry screen scrolls. You'll get these advantages automatically; you can also use a VirtScreen explicitly for other special effects, such as scrolling a large screen smoothly within a smaller window.

AbstractFrame, as the name implies, is an abstract object, which means that you should never create an instance of one. The purpose of such an object is to provide a generalized definition of a concept without implementing any but the barest essentials. An AbstractFrame includes data fields for the coordinates of the frame, its minimum and maximum dimensions (used during resizing) and its position limits (used during moving). The coordinates are all stored as Words, so as not to preclude the possibility of a graphics mode frame. AbstractFrame is provided as a hook for future object-oriented extensions of the window hierarchy; you probably won't need to be concerned about it for now.

The Frame object fills in the details of the AbstractFrame. Like the rest of **Object Professional**, it is optimized for text mode and thus includes details for line-drawing border characters and video attributes. It also contains three linked lists (of headers, shadows, and hot spots) and one array (of pointers to scroll bars). Much of the look and feel of an application is controlled by the appearance of each window's frame. As such, the Frame methods used for

adding and modifying border, header, shadow, and hot spot elements may play an important role in the design of your application. OOP provides a great advantage here: once you've learned the frame customization process for one window object type, you'll know how to do it for all of them, whether you're working on a data entry screen, a menu, an editor, or your own specialized window type.

In spite of the usual connotations associated with a Frame, you don't need to think of it as always drawing a border around a window. Frames can be borderless or bordered. Borderless frames can still have headers, shadows, and hot spots if that's desirable.

AbstractWindow is another abstract object, defining the basic properties of any window. It includes fields for the coordinates, minimum and maximum allowable size, position limits, and last state before zooming. Again, the coordinates are stored in Word variables so as not to preclude a graphics-mode implementation. Most of the methods in AbstractWindow must be overridden; nevertheless, by their very existence they promise that descendants of the AbstractWindow will offer standard capabilities to draw, erase, move, resize, clear, select, and zoom themselves, among others. The AbstractWindow also contains hooks to the AbstractFrame. Each AbstractWindow may have two frames associated with it, one that is displayed when the window is the current window, and another when the window is on-screen but not the current one.

The RawWindow is the most primitive window type that you'll actually instantiate. It is a text mode window, with the following properties:

- screen coordinates
- cursor position and type
- mouse cursor position and type
- default video attributes
- default background character
- ScreenRect for what the window covers
- optional ScreenRect for what the window contains
- various option flags
- main Frame for when window is current
- alternate Frame for when window is visible but not current

When a RawWindow is displayed, the region of the screen it will cover is first saved in a ScreenRect. The current state of the hardware cursor, mouse cursor, default text attribute, and OPCRT window coordinates is stored. The window's previous contents are restored, if available, or a descendant's method is called to update the contents. The RawWindow's main Frame is then drawn. Finally the cursor is positioned, the default text attribute is set, and OPCRT and mouse window coordinates are defined. Subsequent window-relative writes (using the WriteLn statement, for example) will thus be positioned within the RawWindow's boundaries.

When the RawWindow is erased, its cursor position is saved, the current contents of the window are optionally copied to a ScreenRect, the underlying screen is restored, and the cursor position, window coordinates, etc. are set to their previous values.

The RawWindow offers many other methods for actions such as:

- clear or scroll window contents
- write to itself (centered, justified, at specified position)
- specify alternate open and close routines for explosions and other special effects
- position cursor and change shape

- evaluate cursor or mouse position relative to window and frame
- evaluate and update scroll bars
- resize, move, zoom, or arbitrarily adjust window coordinates
- load and store in a stream

The StackWindow works like a RawWindow, with the addition that each StackWindow is automatically pushed onto the global WindowStack (wStack) when it is displayed. Because StackWindows are managed in this way, they can be accessed randomly: any window can be brought to the top of the stack or written to even when not at the top of the stack. The additional data space used by a StackWindow compared to a RawWindow is minimal (12 bytes per instance). Hence, the StackWindow should be used instead of the RawWindow for most applications. All of **Object Professional**'s higher level window types are derived from the StackWindow.

SubMenu is another type derived from RawWindow. It is used within the Menu object, which manages its own stack of the active submenus of a menu system. The SubMenu adds a MenuItemList to the normal data fields of a RawWindow.

CommandWindow adds a new facet to the StackWindow's behavior. It associates a CommandProcessor with the window in order to map keyboard sequences and mouse clicks to logical commands (see Chapter 3). The CommandWindow stores the last command and keystroke entered by the user for reference by the program. It also connects the window to a standard error handling procedure which is called whenever an error occurs within the window. CommandWindow's most commonly used methods are Process and ProcessAutoPilot. Process gives keyboard and mouse control to the window so that it can interpret commands until an exit command is received (or an error occurs). ProcessAutoPilot does the same thing, but runs an arbitrary sequence of commands automatically before returning control to the keyboard.

You probably won't declare instances of type CommandWindow, since it just sets the stage for useful implementations derived from it, such as the PickList, Menu, EntryScreen, and others. If you need to derive other specialized window types requiring keyboard input (a "radio button" window, perhaps), then you should be aware of CommandWindow.

From an object-oriented programming standpoint, a Menu is a particularly interesting derivative of a CommandWindow. If one considers a single level menu system, the correspondence is clear: the Menu simply associates a MenuItemList with a CommandWindow and then manages the selection of a menu item from the list. For multi-level menu systems, the situation gets more complicated, at least from the perspective of the Menu object. The Menu object includes a PointerStack, which manages a stack of SubMenus, which are themselves derived from RawWindows. The rest of the program views this collection of windows as a single one, which can be displayed and moved collectively and which is either active or not.

Because a menu system definition requires so many details–coordinates, item names, border characters, headers, video attributes, etc., for each submenu–we've provided a powerful utility program which lets you design a menu and then writes the source code for you. It's called MAKEMENU and described in section H of this chapter (in Volume 2).

A PickList displays an arbitrary list of items within a CommandWindow and manages the selection of one or more items from that list. The item strings themselves are supplied by a user-defined routine (or by a virtual method of an object type derived from PickList), so that a variable number of items of any predefined width may be used. PickList draws the items in one or more columns using any of three orientations. The PickList offers plenty of flexibility by supporting temporarily protected items, multiple video attributes, items drawn using FlexWrite,

user hooks during the selection process, specialized frame titles to indicate more items off-screen, alphanumeric searching, multiple selections, and mouse support.

The PickList turns out to be a particularly extensible object. Many user-interface tasks can be solved by making a selection from a list. In the DESKPOP demonstration program, for example, the file manager, calendar, and phonebook are all objects derived from the PickList. The MAKEMENU and MAKESCRN utilities use PickList derivatives for selecting valid frames, characters, window headers, and several other categories of items.

In fact, it would be possible to derive the Menu object from the PickList. We've chosen not to do so for at least two reasons: first, we wanted to maintain some degree of backward compatibility with Turbo Professional menus; second, the Menu is more efficient for typical menu systems which contain static one-dimensional lists of items. Of course, the Menu includes additional code to support multi-level, nested menu systems as well. By contrast, the PickList is better suited when the items aren't known in advance, when two-dimensional lists are needed, when there are more items than will fit in one window (scrolling is required), or when selection of more than one item is required at a time. You'll want to consider the relative merits of Menu and PickList whenever a menu-based interface is needed.

DirList and PathList are derivatives of PickList that help to manage the DOS file system. DirList displays lists of files; PathList, lists of subdirectories. Getting your end-user to enter a filename or subdirectory is a snap with the help of these objects. Because the DirList supports selection of multiple files, writing a powerful file manager is easy as well; DESKPOP shows one such example.

The final three objects discussed in this chapter--AbstractHelpWindow, PagedHelpWindow, and ScrollingHelpWindow--are closely related to each other. Each one presents context-sensitive help within a pop-up window. You may be surprised to note that these are derived from the PickList type and not from the more general CommandWindow. The reason is that the help window can display an index of help topics; this presents yet another opportunity for a pick list. Combining this with the capability to display text means that the help window is a dual-mode object, sometimes acting like a pick window and other times more like a file browser.

AbstractHelpWindow is an abstract type that implements the methods used to read help text from an indexed, compressed file of information. It does not implement a way to display the help text. PagedHelpWindow provides one set of methods to display help text while ScrollingHelpWindow offers another. With the paged technique, <PgDn> and <PgUp> bring a complete new page of help into view while the normal arrow keys move a highlight bar from one hypertext link to another. (More on these hypertext links later.) By contrast, for the ScrollingHelpWindow the arrow keys move the cursor around the help window just like in an editor, and the help text scrolls vertically and horizontally one row or column at a time. PagedHelpWindow is backward compatible with Turbo Professional's help system; ScrollingHelpWindow is often preferable in an environment of resizeable windows.

To support the creation of help files we provide a help compiler, MAKEHELP, which converts a text file describing the various help topics into a compressed, random-access file which the window objects can read. Both PagedHelpWindow and ScrollingHelpWindow use the same input format.

That completes the overview of the objects described in this chapter. Further general information, working example programs, and detailed descriptions of data types and methods are provided for each object in the sections that follow.

It's especially important to remember the role of inheritance in the window object hierarchy. All of the customization features, low-level hooks, and general methods of the most primitive window objects remain available to the higher level windows. Use the Procedure and Function Reference or the POPHELP database to help remember all the tools you have at hand.

B. OPFRAME: Frame Objects

OPFRAME defines a number of important object types, but it is unlikely that you'll ever explicitly declare a variable using any of them. These types form a kind of Chinese puzzle where one type fits inside of the other, inside of another, until the largest one, the Frame itself, fits inside of the RawWindow object. A Frame without an associated window will be a rare beast. Even so, customizing a window's Frame requires you to understand and call methods of the Frame object. We'll describe those methods here.

Also included in the OPFRAME unit is the ScreenRect object, because shadows depend on it. The ScreenRect is a low-level object that is nevertheless such a fundamental building block that we will describe it in detail. A ScreenRect provides all the tools needed to save and restore portions of the screen. The RawWindow makes significant use of this object.

Before proceeding, let's make clear the distinctions between some words that we'll use frequently. A *border* is a group of visible line drawing characters that surround a window or frame. A *frame* is a logical data structure associated with a window. The frame contains the characters needed for the border and also includes headers, shadows, scroll bars, and hot spots. A frame may or may not have a visible border. A *window* includes one or more frames, as well as the structures needed to save the underlying screen, the cursor position, the mouse position, and so on.

1. Screen Rectangles

The ScreenRect object is a primitive one, but not so primitive as its non-object-oriented predecessors. One of the first things that developers of early popup window systems added was the ability to save and restore portions of the screen that the window would overwrite. The ScreenRect encapsulates the data and code for these operations so that the same capabilities can be reused for windows and also for similar structures like shadow rectangles. ScreenRect also knows how to resize itself and transfer any previous contents to the new buffer. This is an important service to provide to window objects in a modern mouse-driven environment.

If you are developing new window types, ScreenRect may be of direct use to you; if not, just be aware of its existence.

Types

```
ScreenBuffer      = array[0..65520] of Byte;
ScreenBufferPtr   = ^ScreenBuffer;
```

Used to access buffers that store screen contents in the ScreenRect object.

```
ScreenRect =
  object(Root)
    ...
  end;
```

A low-level object used to store or restore rectangular regions of the screen. It is used internally by many parts of the window system. The ScreenRect contains a pointer to a memory buffer that it allocates from the heap or accepts from the caller.

Adjust

Declaration

```
procedure Adjust(X1, Y1, X2, Y2 : Byte);
```

Purpose

Adjust buffer coordinates to X1, Y1, X2, Y2.

Description

This method may be used to change the screen position of a ScreenRect, change its size, or both. If the size remains unchanged, Adjust doesn't allocate a new buffer. (In particular, if the buffer was stored in some special location other than the heap, it will remain there.) If the buffer size changes at all, Adjust deallocates the old buffer (if it was previously allocated) and allocates a new buffer on the heap. If there is insufficient heap space for the new buffer, Adjust stores the error code epFatal+ecOutOfMemory, which may be checked by calling srResult.

Adjust does not transfer the contents of the old buffer to the new one or initialize the new buffer in any way. Use the Transfer method if that is required.

Example

```
SR.Alloc(1, 1, 10, 10);
SR.Clear(' ', TextAttr);
SR.Adjust(1, 1, 5, 5);
```

Resizes SR from the original 10 by 10 block to a 5 by 5 block. The contents of SR's buffer are lost.

```
SR.Alloc(1, 1, 10, 10);
SR.Clear(' ', TextAttr);
SR.Adjust(20, 1, 29, 10);
```

Shifts SR to new coordinates. Because the size didn't change, the buffer's contents remain intact.

See Also

Alloc
Transfer
srResult

Declaration

```
constructor Alloc(X1, Y1, X2, Y2 : Byte);
```

Purpose

Initialize coordinates and allocate a screen buffer.

Description

Alloc uses a GetMem call to allocate 2*(X2-X1+1)*(Y2-Y1+1) bytes of heap space. This buffer will be used to hold an image of the screen (including both characters and video attributes) from the rectangle whose corners are at (X1, Y1) and (X2, Y2), inclusive. The coordinates are specified in absolute terms, with the upper left corner of the screen being (X=1, Y=1). Alloc sets an internal flag indicating that it allocated the buffer, so that the Done method will know to deallocate it.

If Alloc fails, an error code is stored in the global variable InitStatus.

Alloc does not actually store the screen image. The screen buffer is left uninitialized.

Example

```
uses
  OpCrt, OpFrame;
var
  SR : ScreenRect;
  Row, Col : Integer;
begin
  {Make a screen pattern for the demonstration}
  FastFill(4000, #176, 1, 1, TextAttr);

  if not SR.Alloc(1, 1, 20, 3) then begin
    Writeln('Failed to Allocate ScreenRect');
    Halt;
  end;

  {Move a message around until KeyPressed}
  repeat
    Row := 1+Random(21);
    Col := 1+Random(59);
    SR.Adjust(Col, Row, Col+19, Row+2);
    SR.CopyFromScreen;
    FastText('  Hello World  ', Row+1, Col+1);
    Delay(200);
    SR.CopyToScreen;
  until KeyPressed;
  SR.Done;
end.
```

Allocates a ScreenRect 20 columns wide and three rows high. This ScreenRect is used to repeatedly store the contents of the screen (with CopyFromScreen), then restore the contents (with CopyToScreen) after ' Hello World ' is written there. The Adjust method changes the coordinates of an existing ScreenRect without transferring the old contents (which is unnecessary in this case).

See Also

Init

BufSize / Clear / Coordinates

Declaration
```
function BufSize : Word;
```

Purpose
Return buffer size in bytes.

Description
Returns the product 2*(X2-X1+1)*(Y2-Y1+1), where the variables represent the current coordinates of the ScreenRect. This is the number of bytes used by the screen buffer.

Example
```
SR.Alloc(1, 1, ScreenWidth, ScreenHeight);
WriteLn('This object is using a buffer of ', SR.BufSize, ' bytes');
```
Reports on the size of the buffer allocated by Alloc. For a full screen size of 80x25, BufSize is 4000 bytes.

Declaration
```
procedure Clear(FChar : Char; FAttr : Byte);
```

Purpose
Clear with character and attribute.

Description
Clears the screen rectangle's buffer, much as a ClrScr command would do for a window. FChar is typically a space character, and FAttr is any video attribute. The effect is not visible until the ScreenRect is copied back to the screen.

Example
```
SR.Clear(' ', TextAttr);
```
Clears the ScreenRect display buffer with spaces and the default video attribute.
```
SR.Clear(#178, $1E);
```
Clears the ScreenRect display buffer by filling it with a background pattern in yellow on blue.

Declaration
```
procedure Coordinates(var X1, Y1, X2, Y2 : Byte);
```

Purpose
Return coordinates of screen rectangle.

Description
Provides access to internal data fields of the ScreenRect that store its screen coordinates.

Example
```
var
  Xlo, Ylo, Xhi, Yhi : Byte;
...
SR.Coordinates(Xlo, Ylo, Xhi, Yhi);
WriteLn('Upper left corner:  ', Xlo, ',', Ylo);
WriteLn('Lower right corner: ', Xhi, ',', Yhi);
```
Displays the physical screen coordinates of the specified screen rectangle.

Declaration
```
procedure CopyFromScreen;
```

Purpose
Copy from screen to buffer.

Description
Initializes the screen buffer by copying the current contents of the screen. The most recent set of coordinates passed to the ScreenRect determines what portion of the screen is copied.

When CopyFromScreen reads from the screen, it uses the virtual video segment. Hence, if a virtual screen is active at the time, CopyFromScreen will read the virtual screen's contents instead of the physical screen's.

Example
```
SR.Alloc(1, 10, 20, 12);
SR.CopyFromScreen;
FastText('  Hello World  ', 11, 1);
Delay(500);
SR.CopyToScreen;
```

Copies a portion of the physical screen to SR's buffer. Displays a message within the same rectangle, pauses for 1/2 second, then copies SR's buffer back to the physical screen (restoring the original contents).

See Also
CopyToScreen VirtScreen.Activate

Declaration
```
procedure CopyPartToScreen(X1, Y1, X2, Y2 : Byte);
```

Purpose
Copy part of the buffer to the screen.

Description
X1, Y1, X2, Y2 are absolute screen coordinates. They must lie within the current coordinates of the screen rectangle or CopyPartToScreen will generate an error (epNonFatal + ecWinCoordsBad) and exit. The specified portion of the screen buffer is copied back to the physical screen.

Example
```
SR.Alloc(1, 1, 10, 10);
SR.Clear(' ', TextAttr);
SR.CopyPartToScreen(1, 1, 5, 5);
```

Allocates a 10 by 10 block and clears the ScreenRect buffer; copies only a 5 by 5 portion of that buffer to the physical screen.

See Also
CopyToScreen

CopyToScreen / Done

Declaration

```
procedure CopyToScreen;
```

Purpose

Copy buffer to screen.

Description

The entire screen rectangle is written to its original position on the physical screen. This will typically have the effect of restoring the screen to its state at the time CopyFromScreen was called.

When CopyToScreen writes to the screen, it uses the virtual video segment. Hence, if a virtual screen is active at the time, CopyToScreen will write to the virtual screen instead of the physical screen.

Example

```
SR.Alloc(1, 10, 20, 12);
SR.CopyFromScreen;
FastText('  Hello World  ', 11, 1);
Delay(500);
SR.CopyToScreen;
```

Copies the contents of the physical screen to SR's buffer. Displays a message at those coordinates, pauses for 1/2 second, and copies SR's buffer back to the physical screen, restoring the original contents.

See Also

CopyPartToScreen

Declaration

```
destructor Done; virtual;
```

Purpose

Deallocate buffer.

Description

The ScreenRect keeps track of who allocated its screen buffer. If the ScreenRect itself did so (usually via the Alloc constructor), Done will deallocate the buffer with this call. Otherwise it is the caller's responsibility to deallocate it if necessary.

Example

```
SR.Done;
```

Disposes of the screen rectangle. If the constructor allocated any heap space, that heap space will be disposed.

See Also

Alloc Init

Declaration

```
function HaveContents : Boolean;
```

Purpose

Return True if contents were loaded by Load.

Description

This method has meaning only in relation to streams. When a ScreenRect is stored to a stream, the contents of its buffer can optionally be written to the stream as well. If so, when the stream is later read, the buffer will be initialized from the stream. In this case, HaveContents will return True.

Example

```
SRS.Init('SCRRECT.STM', SOpen, 1024); {Open stream containing a ScreenRect}
if not SR.Load(SRS) then              {Load the ScreenRect}
  Halt;
SRS.Done;                             {Close the stream file}
if not SR.HaveContents then           {Check for contents}
  SR.Clear(' ', TextAttr);            {Clear the buffer if none loaded}
SR.CopyToScreen;                      {Copy buffer to screen}
```

If the ScreenRect just loaded did not have its contents stored in the stream, clear the buffer. Then copy the buffer to the physical screen.

See Also

StoreContents Load

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; BufPtr : Pointer);
```

Purpose

Initialize coordinates and assign a preallocated buffer.

Description

Works like the Alloc constructor except that the buffer address is passed to Init. BufPtr must point to a region of at least 2*(X2-X1+1)*(Y2-Y1+1) bytes.

BufPtr can point to a block on the normal heap or to another region such as an unused video page. Use OPCRT's GetPagePointer routine for the latter option.

Init clears an internal flag so that the Done method will not attempt to deallocate the space pointed to by BufPtr.

Init does not actually store the screen image. The screen buffer is left uninitialized.

Example

```
var
  MyBuf : Pointer;
...
GetMem(BufPtr, 2*ScreenWidth*ScreenHeight);
SR.Init(1, 1, ScreenWidth, ScreenHeight, BufPtr);
```

Get a buffer large enough to store the entire screen; instantiate a screen rectangle using that buffer.

See Also

Alloc GetPagePointer (OPCRT)

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load screen rectangle from stream.

Description

Reads the coordinates of the screen rectangle from the stream. It then reads the pointer code for the screen buffer. If the pointer was registered with the stream, the physical pointer is assigned to the ScreenRect's buffer pointer and no heap space is allocated. (This would be useful if an alternate video page was to be used for the screen buffer.) Otherwise, Load allocates a buffer on the heap.

If the contents of the ScreenRect were stored with it, Load reads and decompresses them into the buffer. HaveContents will return True in this case.

The stream registration procedure for a ScreenRect is ScreenRectStream.

Examples

```
var
  SR : ScreenRect;
  SRS : BufIdStream;
  Status : Word;
...
SRS.Init('SCRRECT.STM', SOpen, 1024);
SRS.RegisterHier(ScreenRectStream);
SRS.Get(SR);
Status := SRS.GetStatus;
if Status <> 0 then begin
  WriteLn('Load error ', Status);
  Halt;
end;
SRS.Done;
```

Opens a stream, registers the ScreenRect type, and initializes SR by calling the stream's Get method, which calls Load indirectly. If the stream contained the contents of the ScreenRect, it can immediately be copied back to the screen using CopyToScreen.

```
...
SRS.Init('SCRRECT.STM', SOpen, 1024);
SRS.RegisterHier(ScreenRectStream);
SRS.RegisterPointer(2000, GetPagePointer(1));
SRS.Get(SR);
...
```

Uses the second video page (of a CGA adapter) for the ScreenRect's buffer, assuming that the stream was stored using code 2000 for that pointer.

See Also

GetPagePointer (OPCRT)
HaveContents
Store

Declaration

```
function srResult : Word;
```

Purpose

Return most recent status code.

Description

Use srResult to check for errors that may occur within certain ScreenRect operations (other than constructors). srResult returns 0 if no error occurred.

The following errors are possible:

```
epNonFatal+ecWinCoordsBad   CopyPartToScreen
epFatal+ecOutOfMemory       Adjust, Transfer
```

Calling srResult resets the internal status variable. Calling it again immediately will always return 0.

Example

```
SR.Transfer(11, 11, 15, 15, ' ', TextAttr);
Status := SR.srResult;
if Status <> 0 then begin
  WriteLn('Failed to Transfer, Status = ', Status);
  Halt;
end;
```

Checks for errors after a call to Transfer.

```
if srResult <> 0 then
  WriteLn('Out of memory ', srResult);
```

Incorrect! Referencing srResult also clears it to zero. Move the value to a variable if you need to reference it more than once.

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store ScreenRect to stream.

Description

First writes the coordinates of the screen rectangle to the stream, then writes the pointer code for the screen buffer. If the ScreenRect was initialized with Init (to use a preallocated buffer), it is generally desirable to register a unique code for the associated pointer. In this way, the Load method can later detect that a buffer should not be allocated. If no pointer code is registered, Store will write the code for a nil pointer, which Load will later interpret as a request to allocate heap space for the buffer.

To store the contents of the ScreenRect in the stream, call StoreContents with a True parameter before calling Store. Store writes the contents in a packed format similar to that used by the PackedWindow object. The characters and attributes of the buffer are separated and then run length encoded. Any run of more than four equal bytes is packed, saving the number of bytes by which the run exceeds four. ScreenRect.Store imposes one restriction that a PackedWindow doesn't, however: it cannot pack a buffer containing the $FF byte, since this byte is reserved as a pack group identifier. If such a character or video attribute is encountered, Store generates a stream error (epNonFatal+ecInvalidData).

Examples

```
var
  SR : ScreenRect;
  SRS : BufIdStream;
  Status : Word;
...
SR.StoreContents(True);
...
SRS.Init('SCRRECT.STM', SCreate, 1024);
SRS.RegisterHier(ScreenRectStream);
SRS.Put(SR);
Status := SRS.GetStatus;
if Status <> 0 then begin
  WriteLn('Store error ', Status);
  Halt;
end;
SRS.Done;
```

Creates a stream file, registers the ScreenRect type, and writes the ScreenRect SR, including its contents, to it. (Put indirectly calls ScreenRect.Store as a result of the registration process.)

```
...
SR.Init(1, 1, ScreenWidth, ScreenHeight, GetPagePointer(1));
...
SRS.Init('SCRRECT.STM', SCreate, 1024);
SRS.RegisterHier(ScreenRectStream);
SRS.RegisterPointer(2000, GetPagePointer(1));
SRS.Put(SR);
...
```

Registers the second video page (of a CGA adapter) with the stream manager so that Load can later use the same area for storage.

See Also

Load StoreContents

Declaration
```
procedure StoreContents(On : Boolean);
```

Purpose
Specify whether Store writes contents of screen buffer.

Description
Calling this method with On equal to True sets an internal flag; Store will write the screen buffer's contents to the stream if the flag is set.

See Also
Store

Declaration
```
procedure Transfer(X1, Y1, X2, Y2 : Byte; FChar : Char; FAttr : Byte);
```

Purpose
Transfer buffer to new buffer at coordinates X1, Y1, X2, Y2.

Description
Use this method to shift or resize the coordinates of an existing ScreenRect, and to transfer the existing contents to the new position. What was in the upper left corner of the previous position will be in the upper left corner of the new position. If the new rectangle is longer in either dimension than the old, the new space will be filled with FChar and FAttr.

If the new coordinates specify a rectangle just the same size as the old, the existing screen buffer is left unchanged (no real transfer is required). Otherwise, Transfer must allocate a new buffer and transfer the old contents to it before deallocating the old. Note that the latter operation is incompatible with maintaining a preallocated screen buffer.

If there is insufficient heap space to allocate the new buffer, Transfer stores the error code epFatal + ecOutOfMemory. Check status by calling srResult.

Examples
```
SR.Alloc(1, 1, 10, 10);
SR.Clear(' ', TextAttr);
SR.Transfer(11, 11, 15, 15, ' ', TextAttr);
```
Transfer allocates a new 5 by 5 block and copies the contents of the original 10 by 10 block to it (the part that will fit, that is).
```
SR.Alloc(1, 1, 10, 10);
SR.Clear(' ', TextAttr);
SR.Transfer(1, 1, 20, 20, ' ', TextAttr);
```
Allocates a new 20 by 20 block and copies the contents of the original 10 by 10 block. The new part of the screen rectangle is filled with blanks in TextAttr.

See Also
Adjust srResult

2. Frames

Let's start out with an example program to get a feel for how frames are used.

```
program FrameExample;  {EXFRAME.PAS}
uses
  OpCrt, OpString, OpRoot, OpFrame, OpWindow;
var
  Win : RawWindow;
  WinOpts : LongInt;
  I, SliderPos : Byte;
  Finished : Boolean;
begin
  {Make a framed, resizable RawWindow}
  WinOpts := wBordered+wClear;
  if not Win.InitCustom(30, 5, 50, 15, DefaultColorSet, WinOpts) then begin
    Writeln('Failed to initialize RawWindow, Status ', InitStatus);
    Halt;
  end;

  {Add some Frame features}
  with Win.wFrame do begin
    SetFrameType(DblWindowFrame);
    AddShadow(shBR, shOverWrite);
    AddHeader('Top Header', heTC);
    AddHeader('Bottom Header', heBC);
    AddScrollBar(frRR, 1, 9, DefaultColorSet);
  end;

  {Draw the window, fill it with text}
  Win.Draw;
  for I := 2 to 10 do
    Win.wFastText('Hello Mars', I, 5);

  {Update window/slider based on up/down cursor commands}
  SliderPos := 1;
  Finished := False;
  repeat
    {Update Win and ScrollBar}
    Win.DrawSlider(frRR, SliderPos);
    Win.wGoToXY(3, SliderPos+1);
    case ReadKeyWord of
      $4800 : if SliderPos > 1 then     {Up arrow}
                Dec(SliderPos);
      $5000 : if SliderPos < 9 then     {Down arrow}
                Inc(SliderPos);
      $011B : Finished := True;         {Escape}
    end;
  until Finished;
  Win.Done;
end.
```

The frame we'll discuss is actually a data field within the RawWindow object Win. The example program first initializes the RawWindow using the InitCustom constructor. A LongInt parameter to this constructor specifies a bitmapped set of window options. One of those options is wBordered, which tells the window to reserve one additional row and column all the way around for the border. Note how the coordinate system is defined: the active window (where the window contents are written) extends from the absolute screen columns X=30 to X=50 and from rows Y=5 to Y=15, as specified to InitCustom. The border of the window is

displayed one position farther out: from X=29 to X=51 and Y=4 to Y=16. We'll discuss the InitCustom constructor in more detail with the RawWindow object.

The most relevant portion of this example appears in the next block, which starts as "with Win.wFrame do begin." wFrame is a data field of type Frame, stored in each RawWindow object. The with statement therefore lets us access the methods of the wFrame object and to customize this window frame as needed. (Each RawWindow contains another frame object, called aFrame, which is used in other circumstances. We'll discuss the distinction between wFrame and aFrame later, in the section for RawWindow.)

The example shows four of the ways to customize a frame. First, SetFrameType specifies the characters used for the border. DblWindowFrame is a constant defined in the OPFRAME unit to mean the double-line IBM high ASCII characters. OPFRAME interfaces additional standard sets of border characters described later. Each array of border characters contains 8 elements, thus providing complete control over the appearance of the border. Without a call to SetFrameType, the window would get the default border, which uses the characters stored in the constant DefWindowFrame.

AddShadow enables the display of shadows around the frame. Shadows lend a three-dimensional appearance and are increasingly popular in today's applications. Shadows in the Frame object are implemented as a linked list of rectangular regions (ShadowNodes). When a frame is displayed, the screen contents beneath each rectangle are stored in a buffer, and each rectangle is drawn to provide the appearance of a shadow. Because of this general approach, shadows are extremely flexible in the Frame object--you could even have shadows on all four sides of a frame if desired. Specifying such non-standard shadows is possible but tedious; the AddShadow routine makes it easy to enable the most common shadow types.

The first parameter to AddShadow, shBR, indicates that the shadows should appear on the bottom and right edges of the window. Another possibility for this parameter is shBL, for shadows on the bottom and left edges of the window. The second parameter, shOverWrite, indicates that the shadow will overwrite existing characters on the screen using another character and video attribute. The alternative here is shSeeThru, which causes the shadow just to change the video attribute, allowing the underlying characters to show through.

Headers are text strings displayed within the frame. These are also implemented as a linked list (of HeaderNodes). As a result, you can have any reasonable number of headers (up to 255) for a given frame. There are several ways to specify headers; the easiest to use is the AddHeader method. AddHeader takes the text string to display as the first parameter and a relative position indicator as the second. heTC means to center the header in the top bar of the frame; heBC means the bottom center; other options allow left and right justified headers.

Scroll bars are usually associated with mouse support, but they may also be used simply as status indicators, as done in this example. A scroll bar consists of four components: the bar's background, which when selected with a mouse causes the window to scroll according to the mouse position; the slider character, which overwrites one position of the background to indicate the current relative position; the decrement arrow, which when selected with a mouse causes a user-defined "decrement" action to occur; and the increment arrow. The characters and video attributes associated with each of these components may be adjusted.

Each frame may have up to four scroll bars, one for each edge of the frame. The AddScrollBar method specifies which edge--frRR means the right edge; frLL, frTT, and frBB are the others.

Each scroll bar is associated with a range of user values. For example, the top of a scroll bar might mean line 1 in a text file and the bottom of the bar line 1000, the last line. When the

current line in the text file is line 1, the scroll bar's slider character will then be at the top of the bar. As the current text line is increased, the slider moves proportionally down the bar until it reaches the bottom at line 1000. The second two parameters to AddScrollBar specify the range of user values that corresponds to the full range of the scroll bar. These parameters are of type LongInt, thus allowing any discrete numeric type for the user range.

The final parameter to AddScrollBar is a ColorSet. AddScrollBar grabs the requisite colors for the bar's background, slider character, and increment/decrement arrows from this set.

AddScrollBar internally uses the fifth main component of a frame: hot spots. Each hot spot specifies an invisible rectangular region of the screen. Associated with each one is a unique numeric code. Methods are available to classify a given screen coordinate (usually a mouse position) with respect to the hot spots and return the unique code if the position falls within a spot. Hot spots are managed in a linked list, so each frame can have any number of them. For each scroll bar, three hot spots are automatically added: one each for the increment and decrement arrows, and another for the background region. In this example we don't take advantage of the hot spots.

The example program displays the window, writes some text to it, and then reads keys until <Esc> is pressed. The most important thing to note here is that the frame is drawn automatically. Once you've added the frame features you don't need to worry much about them. The only remaining reference to the frame is via the call to DrawSlider. Here we update the scroll bar to indicate a new user position contained in the variable SliderPos.

The example ends with a call to Win.Done. The Done destructor not only deallocates the window itself but also gets rid of the frames it contains. After this call the linked lists of headers, shadows, and hot spots, as well as the scroll bars, have been deallocated.

We'll now cover some frame-related topics in greater detail.

Conditional Defines

OPFRAME is a very full-featured frame manager. Providing this flexibility requires a substantial amount of object code (about 20,000 bytes in the entire OPFRAME unit). Turbo Pascal's smart linker can strip much of the unused code from a particular program, but it cannot strip all such code, especially in the presence of virtual methods.

To control code size (and to a lesser extent, data size) **Object Professional** counts on a number of conditional defines, all of them managed via the source file OPDEFINE.INC. Several of these defines apply to OPFRAME and the units that depend on OPFRAME. You'll want to disable as many of the defines as you can, consistent with the needs of your program. By convention, defines are disabled by placing a period between the left comment brace and the dollar sign, for example

```
{.$DEFINE UseScrollBars}
```

Defines that affect OPFRAME are as follows:

```
{$DEFINE UseMouse}
{$DEFINE UseScrollBars}
{$DEFINE UseHotSpots}
{$DEFINE UseShadows}
{$DEFINE UseAdjustableWindows}
```

All of these are active by default. When you disable one, certain methods are effectively removed from the source code. These defines also have effects on other units besides OPFRAME, particularly OPWINDOW.

Not all possible combinations of these defines make sense and you shouldn't expect logical program behavior if you pick a particularly bizarre set of options. Typically, if you don't UseMouse, you won't UseScrollBars or UseHotSpots. Similarly, if you don't UseScrollBars, you won't UseHotSpots either. UseShadows is independent of the other defines. UseAdjustableWindows is independent of the others, but is used most frequently when a mouse is available.

In the following documentation, methods and data structures whose existence depends on a particular define are preceded with an {$IFDEF TheDefine}.

Disabling all of these defines can reduce a window-based application's EXE size by over 17K bytes (counting the cumulative effects on OPFRAME and OPWINDOW). In terms of final EXE size, shadows cost roughly 2K bytes; mouse, scroll bars and hot spots (together) cost 8K bytes; and adjustable windows cost 7K bytes,

How a Frame object is written to a stream depends on which conditional defines are active. When a stream was written with one set of defines, you must read it with exactly the same set of defines.

Using Window Headers

The most common use of a window header is to provide a title string centered in the top segment of the window frame. This is easy to do with OPFRAME. However, there are many other ways to use headers and we'll discuss some of them here.

As already mentioned, the Frame object manages headers using a linked list of HeaderNodes. Each HeaderNode stores a pointer to the actual text string as well as the color and monochrome video attributes used to display the header. Four data fields within the HeaderNode fully describe the position of the header relative to the frame:

```
heDX    : Integer;       {Frame-relative coordinates for header}
heDY    : Integer;
heType  : HeaderPosType; {Type of header, used when resizing}
hePosn  : FrameCharType; {Portion of frame header is relative to}
```

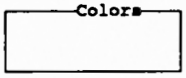
HeaderPosType is an enumerated type that describes the various categories of header positioning:

```
HeaderPosType = (heCustom, heSpan, heTL, heTC, heTR, heBL, heBC, heBR);
```

It's worth describing these positioning options since they embody much of the header's flexibility. The most commonly used options will be heTL..heBR. When you add a header using AddHeader or AddHeaderColor, you initialize a HeaderNode with one of these HeaderPosType values. These describe headers that are placed at standard positions within the frame: at the left edge of the top bar, centered in the top bar, at the right edge of the top bar, and so on for the bottom bar of the frame. If you resize a frame (usually by calling ResizeWindow) the frame manager will correctly adjust the header coordinates so that the title remains in the same relative position.

The heCustom option provides complete flexibility in positioning a header string. The heCustom option is accessed by using the AddCustomHeader method in the Frame object. The rest of the information describing the string's position is stored in the heDX, heDY, and hePosn fields. hePosn specifies the corner of the frame to which the header is relative. It can take on the values frTL (top left), frTR (top right), frBL (bottom left), and frBR (bottom right), which specify the frame's four corners. heDX and heDY specify the distance in columns and rows, respectively, from the specified corner to the start of the header string.

You may have realized that there are four different ways to specify any position within the frame, depending on which hePosn corner is used as the base. The choice becomes important when the window is resized, at which time the header's distance from the chosen corner remains fixed. Suppose for example that you want a header that looks like this:



and that when the window is resized you want the Colors string to move in concert with the upper right corner of the window. To achieve this goal, the values stored in the HeaderNode would be hePosn = frTR, heDX = -9, and heDY = 0.

There's nothing that limits the position of a custom header to within the actual border of the frame. For example, you could place a row of titles inside the window. Because a window's frame is drawn after its contents are cleared or restored, such titles would remain visible as long as your code that writes to the window did not overwrite them.

The heSpan option provides a particularly useful capability called a span header. Such a header spans from one edge of the frame to the other, merging into the frame's border on each edge. These are handy for dividing menus or entry screens into smaller working sections or for separating column titles from the lists they describe. When you call the Frame object's AddSpanHeader method, you specify the first character, the spanning character, and the last character of the span. The first and last characters should be chosen to provide a smooth merge between frame and span. The span may be either vertical or horizontal; this is specified by choosing an edge of the frame to which the span is parallel–frLL (left edge), frRR (right edge), frTT (top edge), frBB (bottom edge). This value is stored within the hePosn field. The combination of edge and offset distance determines the exact position of the span and its behavior when the window is resized.

Any of the Add methods for headers can fail by running out of heap space. Be sure to check for errors by using the Frame.GetLastError function.

So far we've talked of headers as static text; that is, something added when the frame is created and not changed thereafter. It's also possible to change headers on the fly. **Object Professional** does this in various higher level windows: the "Up/Down for More" prompt in a pick list or a pathname/wildcard title in a directory window, for example.

To refer to an existing header, you use a byte variable called its header index. This is simply the sequential position of the header in the linked list of headers, starting with 0. You can count these yourself or you can use the Frame object's GetLastHeaderIndex method to return the index of the last header added. Later you can call Frame's ChangeHeaderString or ChangeHeaderAttr methods or RawWindow's ChangeHeader method with this index in order to modify the header text or attributes and optionally update the display. These details are taken care of automatically when you use the PickList, DirList, PagedHelpWindow, and other objects. You may need to know them if you're deriving specialized window types.

It's also possible to temporarily disable specified headers, so that they are not drawn along with the rest. See the DisableHeader method for more information.

Using Shadows

Shadows offer a level of flexibility similar to headers when customizing a frame. The shadows of a frame are stored as a linked list of ShadowNodes. Since you can add an arbitrary number of ShadowNodes your windows can have shadows on one side, two sides, or even all.

The most convenient method of specifying shadows is the Frame object's AddShadow routine. This method lets you specify a shadow that surrounds the bottom and right edges, or the bottom and left edges of the frame. The shadow may be displayed in either of two ways: shSeeThru or shOverWrite (which are both part of the ShadowDrawType enumerated type).

The shSeeThru variety of shadow displays itself by simply changing the video attribute of the shadow region. With careful choice of attributes, you'll still be able to see the underlying text. The vertical section of this shadow is two columns wide and the horizontal portion is one row high. The attributes used for the shadow are taken from the frShadowColor and frShadowMono fields of the Frame object, which are themselves initialized from the ShadowColor and ShadowMono fields of a ColorSet when the Frame is initialized.

The shOverWrite shadow displays itself by writing a new character and attribute to the shadow region. Various high order characters make good choices for the shadow region. The typed constants DefShadowCharH and DefShadowCharV are used to initialize the horizontal and vertical shadow characters of each ShadowNode at the time AddShadow is called. The vertical section of the shadow is only one column wide for this case.

The AddShadowColor method works like AddShadow but allows specification of a non-default set of video attributes.

The AddCustomShadow method is the most primitive and most flexible technique. With it you specify one rectangular shadow region at a time. Building a shadow like the one created by AddShadow would therefore require two calls to AddCustomShadow. Use this method to specify shadows on non-standard edges of a frame or with different width or height than the default.

Any of the Add methods for shadows can fail by running out of heap space. Be sure to check for errors by using the Frame.GetLastError function.

One further aspect of using shadows deserves mention here. OPFRAME includes logic to clip shadows when they would fall outside of a specified region of the screen. Hence, shadows do not set a limit on the ability to move or resize a window.

Using Scroll Bars and Hot Spots

These objects are the most dynamic of the entities associated with a frame. The scroll bar display must be updated whenever the "current position" changes within the associated window. If the mouse cursor falls within a scroll bar, the position must be interpreted in terms of the user range associated with the scroll bar. When the mouse cursor falls within a hot spot another action may be indicated.

These objects are generally used only when mouse support is designed into an application. Occasionally, scroll bars may be useful as status indicators even when no mouse is available. Where it makes sense, scroll bar support is already integrated into **Object Professional**'s higher level window objects. For example, the PickList automatically supports scrolling via a scroll bar whenever it detects that a bar has been added to its frame.

The AddScrollBar method is the easiest way to create a scroll bar. You specify which of the four frame edges to use for the scroll bar and it computes the appropriate screen coordinates to use. The increment and decrement arrows are positioned at the extremes of the frame edge (without writing to the frame corners themselves) and the background of the bar fills the region between. Characters for the increment and decrement arrows are taken from the appropriate elements of the DefArrows typed constant array (depending on whether the bar is horizontal or

vertical). Characters for the bar slider and background are taken from the DefSliderChar and DefBarChar typed constants.

AddCustomScrollBar works somewhat more flexibly. The exact portion of the frame edge to contain the scroll bar, as well as the characters to form the bar and slider, may be specified.

No matter which Add method is used, the scroll bar must be at least 3 characters long—one each for the increment and decrement arrows and one for the slider. Any less space causes the Add call to fail. OPFRAME's ScrollBar object always requires the increment and decrement arrows; they can be made effectively invisible (by drawing them with the same characters as the underlying frame) and they can be ignored when the mouse selects them, but they always take space.

Either of the Add methods for scroll bars can fail by running out of heap space. Be sure to check for errors by using the Frame.GetLastError function.

How the scroll bar maps to the contents of the frame (or window) is an important concept to understand. The position of the slider character corresponds to a "current position" within the window. The meaning of current position varies depending on the kind of window. If the window contains a text editor, for example, the current position associated with a vertical scroll bar could be the line number where the editor's cursor resides. For a horizontal scroll bar, it could be the number of columns the editor has horizontally scrolled or perhaps the editor's cursor column. The slider character's position is intended to provide a quick estimate of the relative state of the window.

The ScrollBar object linearly maps the user range to the visible range using the following relation:

$$\frac{\text{CurPos} - \text{MinPos}}{\text{MaxPos} - \text{MinPos}} = \frac{\text{CurUser} - \text{MinUser}}{\text{MaxUser} - \text{MinUser}}$$

MinPos and MaxPos correspond to the bar's absolute screen coordinates (the row for a vertical scroll bar, or the column for a horizontal one); CurPos to the slider's position; MinUser, MaxUser, and CurUser to the corresponding user values.

The minimum and maximum values are specified when a scroll bar is added and may be updated as required. The current user value will be updated frequently, perhaps after each command is entered within the window. The example at the beginning of this section demonstrates using a ScrollBar.

Hot spots are a little tricky to design because they are invisible. No indication of a hot spot appears on the screen, so it can be detected only by moving the mouse cursor and attempting selection at various points. Of course, a well-designed application will provide visible text to go with each hot spot, but it's your responsibility to choose the text and align it with the spots.

Each hot spot is mapped to a byte-valued identifying code. When your application detects that the mouse is currently within a hot spot, this code will specify what action the program should take. Certain codes have predefined meanings; the rest are available to you. Here is a list of the codes:

```
hsNone          = 0;        {Not a hot spot}
hsDecV          = 1;        {Decrement arrow of a vertical scroll bar}
hsIncV          = 2;        {Increment arrow of a vertical scroll bar}
hsDecH          = 3;        {Decrement arrow of a horizontal scroll bar}
hsIncH          = 4;        {Increment arrow of a horizontal scroll bar}
```

```
hsBar              = 5;        {Slider portion of a scroll bar}
hsSpot             = 6;        {Single character hot spot}
hsRegion0          = 7;        {User-defined region}
...
hsRegion9          = 16;
...
hsReservedForTP    = 255;      {Used internally}
```

Codes 1-5 are always associated with a scroll bar. Codes 7-16 are for used-defined hot spots. Codes 17-254 are available for any use. Code 255 is reserved.

There are four different ways to add hot spots to a frame. These methods differ in their flexibility and also in the behavior of the resulting spot when the frame is resized.

The AddHotSpot method adds a single character hot spot to one of the frame's four corners. This method is convenient in that it also adds a single character header to the same corner, thus making the hot spot visible. Such single character spots are often used to indicate that the associated window should be resized or closed. The hot spot code returned is always hsSpot. A code for the frame corner is also computed by the EvaluateMousePos method of the RawWindow object.

AddHotBar adds a hot spot that covers the entire edge of a frame. This spot is invisible unless you provide additional headers associated with it. If the frame is resized, the spot grows or shrinks with it. A user-defined code is specified when AddHotBar is called.

AddHotHeader specifies a hot spot to coincide with a standard window header. In combination with a specified length, the hot spot can be centered, left-justified, or right-justified in the top or bottom bar of a frame. DESKPOP has such a hot spot coinciding with the top title of each window; it is used to drag the window around the screen.

AddHotRegion is the most general method. It specifies a spot of arbitrary rectangular dimensions at an arbitrary distance from any frame corner. These dimensions remain fixed when the frame is resized. DESKPOP uses these hot spots for the local menu activator near the upper left corner of each window.

Any of the Add methods for hot spots can fail by running out of heap space. Be sure to check for errors by using the Frame.GetLastError function.

Like a header, each hot spot has an associated index value. By using this index you can refer to an existing hot spot and change its size or position.

Constants

```
hsNone             = 0;        {Not a hot spot}
hsDecV             = 1;        {Decrement arrow of a vertical scroll bar}
hsIncV             = 2;        {Increment arrow of a vertical scroll bar}
hsDecH             = 3;        {Decrement arrow of a horizontal scroll bar}
hsIncH             = 4;        {Increment arrow of a horizontal scroll bar}
hsBar              = 5;        {Slider portion of a scroll bar}
hsSpot             = 6;        {Single character hot spot}
hsRegion0          = 7;        {User-defined region}
...
hsRegion9          = 16;
...
hsReservedForTP    = 255;      {Used internally}
```

Values used to identify hot spots. Such values are specified when adding the hot spot and are returned by the RawWindow.PosResults method after calling RawWindow.EvaluateMousePos or EvaluatePos.

```
NoFrameChar        = #255;
```

When used in a FrameArray indicates that no writing will be done in that position of the frame. The NoWindowFrame constant (described below) is a complete array of NoFrameChar.

```
SeeThruChar        = #255;
```

Character indicates a see-through shadow when passed as a parameter to the AddCustomShadow method.

```
DefWindowFrame     : FrameArray = 'ÉÍ»º¼Í¼ºÈ';
DblWindowFrame     : FrameArray = 'ÖÄ·º½Ä½ºÓ';
SglWindowFrame     : FrameArray = 'Ú Ä¿³ÀÄÙ³';
NoWindowFrame      : FrameArray = NoFrameChar+NoFrameChar+
                                  NoFrameChar+NoFrameChar+
                                  NoFrameChar+NoFrameChar+
                                  NoFrameChar+NoFrameChar;
```

Used to specify common border types to the SetFrameType method. DefWindowFrame is the default border used when creating a frame. If many windows of the same appearance are to be created, it may be convenient to assign a new value to DefWindowFrame before creating the windows.

```
{$IFDEF UseScrollBars}
DefArrows          : ArrowArray = #27#26#24#25; {Left-Right-Up-Down arrows}
TriangleArrows     : ArrowArray = #17#16#30#31; {Alternate, using triangles}
{$ENDIF}
```

Used to specify the appearance of increment/decrement arrows for scroll bars. DefArrows contains the default arrows used when creating a scroll bar with AddScrollBar or AddCustomScrollBar. TriangleArrows is an alternate set of arrows provided for convenience.

```
{$IFDEF UseScrollBars}
DefSliderChar      : Char = #178; {Character for slider}
DefBarChar         : Char = #176; {Character for scroll bar}
{$ENDIF}
```

Used to specify the appearance of the scroll bar background and slider. These default values apply to scroll bars created using the AddScrollBar method.

```
{$IFDEF UseShadows}
DefShadowCharH : Char = #223; {Character for horizontal shadow}
DefShadowCharV : Char = #219; {Character for vertical shadow}
{$ENDIF}
```

Used to specify the appearance of opaque (shOverWrite) shadows. These defaults apply to shadows created with calls to AddShadow and AddShadowColor.

```
frInsideActive = 8;
frInsideFrame  = 9;
frOutsideFrame = 10;
```

These constants continue a sequence of bytes that started in the OPCRT unit. All the constants describe a screen position relative to a particular window frame. The values in OPCRT defined the frame itself--its corners and edges. These additional constants define positions inside and outside of the frame and its associated window. The distinction between frInsideActive and frInsideFrame isn't clear until you consider how a frame and a window work together. The frame and window don't necessarily abut exactly. The window's coordinates determine where the active work goes on, perhaps where a pick list or editor scrolls. In some cases, the frame will be expanded to leave a section of extra screen space surrounding the active window; this is useful for effects like non-scrolling column headers over pick lists. The frInsideFrame constant applies to this in-between space and frInsideActive refers to the active window itself.

These additional constants are returned by the PosResults method of the RawWindow object.

Types

```
{$IFDEF UseScrollBars}
ArrowDirections   = (arrowLeft, arrowRight, arrowUp, arrowDown);
ArrowArray        = array[ArrowDirections] of Char;
{$ENDIF}
```

Data structure describing the arrows used for building scroll bars.

```
AbstractFramePtr  = ^AbstractFrame;
AbstractFrame     =
  object(Root)
    ...
  end;
```

Abstract data type describing a window frame. This object stores the coordinates of the frame (using Words so as not to preclude a graphics mode implementation) and defines some basic access methods for frames. This object is defined primarily for programmers who wish to make fundamental changes and extensions to the **Object Professional** window system. Some methods of the object are documented below since they are not overridden by the more specific Frame implementation.

```
FramePosType      = frTL..frOutsideFrame;
FrameCornerType   = frTL..frBR;
FrameEdgeType     = frTT..frRR;
```

Types that specify various frame-relative positions. FramePosType covers all possible positions (see the discussion of frInsideActive above). FrameCornerType includes only the four corners of a frame and FrameEdgeType only the four edges. These types are used primarily to qualify parameters passed to various methods of the Frame object.

```
HeaderPosType     = (heCustom, heSpan, heTL, heTC, heTR, heBL, heBC, heBR);
```

Categories of window headers. See the section labeled "Using Window Headers" above for further discussion.

```
HeaderPtr = ^HeaderNode;
HeaderNode =
  object(DoubleListNode)
    ...
  end;
```

Data type describing a window header. A linked list of these nodes is contained within the Frame object. You probably won't need to use any of the data fields or methods of the HeaderNode object directly since the Frame object manages them for you.

```
{$IFDEF UseShadows}
ShadowPtr = ^ShadowNode;
ShadowNode =
  object(DoubleListNode)
    ...
  end;
{$ENDIF}
```

Data type describing a window shadow. A linked list of these nodes is contained within the Frame object. Each ShadowNode describes a single rectangular screen region; hence, for a standard two-edge shadow, there will be two nodes on the list. You probably won't need to use any of the data fields or methods of the ShadowNode object directly since the Frame object manages them for you.

```
{$IFDEF UseShadows}
ShadowDrawType    = (shSeeThru, shOverWrite, shNone);
ShadowPosType     = (shBR, shBL);
{$ENDIF}
```

Enumerated types describing how shadows are displayed. A "see-through" shadow is displayed by changing the video attribute of the underlying screen but leaving the characters unchanged. An "overwrite" shadow changes all underlying characters and attributes to

specified values. shNone is returned by certain methods of the Frame and RawWindow objects when no shadows are in use.

ShadowPosType describes the two standard composite shadows available with the AddShadow and AddShadowColor methods. These specify shadow portions on the bottom and right edges, or bottom and left edges, respectively. Other combinations can be created by using the AddCustomShadow method.

```
{$IFDEF UseHotSpots}
HotSpotPtr = ^HotNode;
HotNode =
  object(DoubleListNode)
    ...
  end;
{$ENDIF}
```

Data type describing a "hot spot." A linked list of these nodes is contained within the Frame object. You probably won't need to use any of the data fields or methods of the HotNode object directly since the Frame object manages them for you. See "Using Scroll Bars and Hot Spots" above for more information.

```
{$IFDEF UseScrollBars}
ScrollBarPtr = ^ScrollBar;
ScrollBar =
  object(Root)
    ...
  end;
{$ENDIF}
```

Data type describing a scroll bar. An array of four pointers—one for each frame edge—to these nodes is contained within the Frame object. You probably won't need to use any of the data fields or methods of the ScrollBar object directly since the Frame object manages them for you. See "Using Scroll Bars and Hot Spots" above for more information.

```
FramePtr = ^Frame;
Frame =
  object(AbstractFrame)
    ...
  end;
```

The main object type in the OPFRAME unit. This object contains data fields describing everything about the frame—coordinates, video attributes, border characters, headers, shadows, hot spots, and scroll bars. The methods of this object will be used extensively whenever windows are used. All Frame methods are described below.

Note that certain data fields and methods of the Frame object "disappear" when the conditional defines discussed in this section's introduction are removed. It is important, especially when using streams or libraries, to choose a set of these defines and stick with them.

Declaration

```
procedure AddCustomHeader(S : string; Posn : FrameCornerType;
                          DX, DY : Integer; AttrColor, AttrMono : Byte);
```

Purpose

Add a custom header.

Description

AddCustomHeader provides the maximum degree of flexibility for positioning a header. S is the header string to display. Posn specifies the frame corner to which the DX and DY offsets are relative. DX and DY are the number of columns and rows, respectively, from that corner to the leftmost character of the header string. (Note that DX and DY may be zero or negative if appropriate.) AttrColor and AttrMono are the video attributes to use for color and monochrome video modes, respectively.

Posn may take on the values frTL, frTR, frBL, and frBR. If the header's frame is moved or resized, the header remains fixed at the specified distance from the specified corner.

When the header is displayed and the starting X coordinate of the header is within the frame's bounds, the header will be clipped (if necessary) to stay within those bounds. If the starting coordinate is outside of the frame's bounds, it is clipped only to avoid overflowing the screen edge. If the starting coordinate would fall outside of the screen, nothing will be displayed, even if some part of the header would appear otherwise.

Examples

```
Win.wFrame.AddCustomHeader('Custom Header', frTL, 2, 0, $02, $70);
```

Adds a custom header to Win's main frame. The location of the header will be calculated using the top left corner (frTL) of the frame using an X offset of 2 columns and a Y offset of 0 rows. Thus, the header will start two characters to the left of the frame's top left corner and will be on the top frame bar. The video attribute is green on black for color modes, reverse video for monochrome.

```
Win.wFrame.AddCustomHeader('Custom header', frBR, -20, -1, $02, $70);
```

Adds a custom header to Win's frame 20 columns to the left and 1 row above the bottom right corner (frBR) of the frame.

See Also

AddHeader AddSpanHeader

AddCustomScrollBar

Declaration

```
{$IFDEF UseScrollBars}
procedure AddCustomScrollBar(Posn : FrameEdgeType;
                             MinUser, MaxUser : LongInt;
                             DecOffset, IncOffset : Byte;
                             SliChar, BarChar : Char;
                             var Colors : ColorSet);
```

Purpose

Add a custom scroll bar.

Description

AddCustomScrollBar provides the maximum degree of flexibility for positioning a scroll bar. Posn is the edge of the frame where the scroll bar will appear. Each edge can have at most one scroll bar.

MinUser and MaxUser represent the corresponding user values when the slider is at the extreme ends of the bar. (See the introduction to this section for more information about the mapping between user values and scroll bar values.) In many cases, the user range changes dynamically (for example, when the number of text lines varies in an editor); it's easy to change the range at any time with the ChangeScrollBar method.

DecOffset is the number of characters from the corner of the frame edge to the decrement arrow of the scroll bar. If DecOffset is 0, the decrement arrow will appear in the very corner of the frame; if it is 1, the arrow will appear one character in from the corner. IncOffset provides the same adjustment for the increment arrow. Note that in both cases positive offsets specify the distance from the corner to the arrow. At least three characters must remain after DecOffset and IncOffset are subtracted from the length of a frame edge: one each for the two arrows and another for the slider character. AddCustomScrollBar will generate an error (epNonFatal+ecWinCoordsBad) if less space is available.

SliChar and BarChar are the characters drawn for the slider and the background of the bar, respectively. If you want the bar to be effectively invisible (as was the case with Turbo Professional's scroll bars), set BarChar to the same character as that used for the frame's border.

The arrow characters used are copied from the typed constant DefArrows when AddCustomScrollBar is called. To use non-default arrows, modify this typed constant first. The following table shows the appropriate array element (of the ArrowArray type) to modify for vertical and horizontal scroll bars.

	Vertical	Horizontal
Decrement arrow	arrowUp	arrowLeft
Increment arrow	arrowDown	arrowRight

Colors contains the attributes used for the various components of the scroll bar. The following fields of the ColorSet are used:

Slider	SliderColor, SliderMono
Bar	ScrollBarColor, ScrollBarMono
Arrows	HotSpotColor, HotSpotMono

Besides adding the structures specific to the scroll bar, AddCustomScrollBar adds three hot spots to the frame: one each for the two arrows and another for the slider region. These hot spots are used later to manage the interaction between a mouse and the scroll bar.

If another scroll bar was previously added to the specified frame edge, it is deallocated before the new one is added. If there is insufficient heap space for the new bar (about 30 bytes) or the hot spots, AddCustomScrollBar generates an error (epFatal+ecOutOfMemory).

Examples

```
Win.wFrame.AddCustomScrollBar(frRR, 0, 9, 1, 1, #7, #179, DefaultColorSet);
```

Adds a custom scroll bar to the right edge of the window using custom scroll bar and slider characters. The scroll bar control arrows are located one row from the top and bottom of the window.

```
DefArrows[arrowUp] := #30;
DefArrows[arrowDown] := #31;
Win.wFrame.AddCustomScrollBar(frRR, 0, 9, 1, 1, #7, #179, DefaultColorSet);
```

Uses triangle-shaped arrows instead of the default ones.

See Also

AddScrollBar ChangeScrollBar

Declaration

```
{$IFDEF UseShadows}
procedure AddCustomShadow(ShChar : Char; Edge : FrameEdgeType;
                          DX, DY : Integer; W, H : Byte;
                          AttrColor, AttrMono : Byte);
```

Purpose

Add a custom shadow rectangle.

Description

AddCustomShadow provides the maximum degree of flexibility for creating a shadow. By calling it instead of AddShadow, you can build shadows that look any way you like. A complete shadow is the composite of one or more shadow nodes, where each shadow node is a simple rectangular region. AddCustomShadow adds a single shadow node at a time.

ShChar specifies the character to fill the rectangular shadow region when it is drawn. For a transparent shadow (one where the underlying character is not overwritten, only its attribute is changed), specify the constant SeeThruChar (#255). Edge is the edge to which the shadow rectangle is parallel. For example, to specify a shadow node at the right edge of the frame, specify frRR. DX and DY specify the position of the shadow's upper left corner relative to a frame corner corresponding to Edge. The following corners are used:

Edge	Corner
frLL	frTL
frRR	frTR
frTT	frTL
frBB	frBL

For a standard one-character-wide shadow on the right edge of a frame, DX is 1. DY is 1 to stagger the shadow at the upper right corner of the frame. For a standard shadow on the left edge, DX is -1 and DY is 1.

W and H are the width and height of the shadow rectangle. For a vertical shadow, W is typically 1 or 2 and H is the height of the frame.

AttrColor and AttrMono are the video attributes used to display the shadow in color and monochrome modes.

Example

```
Win.wFrame.AddCustomShadow(#178, frBB, 1, 1, Win.Width+2, 1, $02, $70);
Win.wFrame.AddCustomShadow(#178, frRR, 1, 1, 1, Win.Height+2, $02, $70);
```

Adds shadow bars to the top and right of the window (like a shBR shadow) using a background fill character. We use the functions Height and Width from RawWindow to size the shadow lines.

See Also

AddShadow

AddHeader / AddHeaderColor

Declaration

```
procedure AddHeader(S : string; Posn : HeaderPosType);
```

Purpose

Add a standard header.

Description

AddHeader is the most convenient way to add a header to a frame. S is the string to display. The Frame object does not pad this string in any way. The string may be changed later by calling ChangeHeaderString.

Posn specifies any of six valid positions for the header. They are: heTL, heTC, heTR, heBL, heBC, and heBR, where the first three are in the top bar of the frame and the last three in the bottom. heTL and heBL position the header at the left edge of the frame (one in from the corner); heTC and heBC center it; and heTR and heBR justify it at the right edge (one in from the corner). The frame automatically maintains these positions when the frame is resized.

The header gets the standard header color specified when the frame was created. Specify a different attribute by calling AddHeaderColor instead, or by calling ChangeHeaderAttr later.

AddHeader can fail with an out of memory error since it allocates heap space for a linked list node and for the string itself.

Example

```
Win.wFrame.AddHeader('Standard Header', heTC);
```

Adds a header to Win's frame. The header will be in the top center of the frame (heTC).

```
Win.wFrame.AddHeader('Bottom Left', heBL);
```

Adds a header to the bottom left of Win's frame.

See Also

AddCustomHeader

Declaration

```
procedure AddHeaderColor(S : string; Posn : HeaderPosType;
                         AttrColor, AttrMono : Byte);
```

Purpose

Add a standard header with custom color.

Description

Works just like AddHeader but allows a non-default attribute to be specified. AttrColor will be used whenever the header is displayed in a color video mode; AttrMono in a monochrome mode (2 or 7).

Example

```
Win.wFrame.AddHeaderColor('Standard Header', heTC, $02, $70);
```

Adds a header to the top center of Win's frame. The header color will be green on black for color monitors, reverse video for monochrome monitors.

Declaration

```
{$IFDEF UseHotSpots}
procedure AddHotBar(Posn : FrameEdgeType; Code : Byte);
```

Purpose

Add a hot region covering an entire frame edge.

Description

Posn is the frame edge to cover: frLL, frRR, frTT, or frBB. Vertical hot regions are one column wide, horizontal ones are one row tall. The resulting hot region does not include the corners of the frame unless the frame is borderless. When the frame is resized the hot bar's position and size is adjusted automatically.

Code is a unique byte value used to identify this particular hot region for this frame. It should be given a value between hsRegion0 and 254, inclusive.

Example

```
Win.wFrame.AddHotBar(frLL, hsRegion0);
```

Adds a vertical hot bar to the left frame of the window with a user code of hsRegion0.

See Also

AddHotSpot AddHotRegion

Declaration

```
{$IFDEF UseHotSpots}
procedure AddHotHeader(hType : HeaderPosType; Code : Byte; Len : Byte);
```

Purpose

Add a hot region to coincide with a header.

Description

This method is useful when you want a hot spot that coincides exactly with a visible header. One common example: a header at the top of a frame that specifies the file within the window. Selecting this header with a mouse will indicate that you want to reposition the window. (This technique is used by DESKPOP, which mimics Microsoft Windows and the Presentation Manager.) When the frame is resized the hot header's position is adjusted automatically.

hType is one of the standard header positions: heTL, heTC, heTR, heBL, heBC, or heBR. See AddHeader for more information. Len specifies the string length of the header to which the hot region corresponds.

Code is a unique byte value used to identify this particular hot region for this frame. It should be given a value between hsRegion0 and 254, inclusive.

Example

```
Win.wFrame.AddHotHeader(heTC, hsRegion1, Length(HeaderString));
```

Makes the top header of the window a hot region with a user code of hsRegion1. The width of the header is the same as the length of the associated header string.

See Also

AddHotRegion

AddHotRegion / AddHotSpot

Declaration

```
{$IFDEF UseHotSpots}
procedure AddHotRegion(Posn : FrameCharType; Code : Byte;
                       DX, DY : Integer; Width, Height : Byte);
```

Purpose

Add a fixed size hot region relative to the frame corner.

Description

This method provides the most flexible way to add a hot region. You can specify any fixed size rectangle at a fixed distance from a specified frame corner. This relationship continues to hold after the frame has been resized.

Posn can be any corner or edge of the frame. We recommend that you specify only corners (frTL, frTR, frBL, frBR); edges are used internally in support of other hot spot structures.

DX and DY are the offsets of the upper left corner of the hot region from the specified frame corner. Note that these can be positive or negative. Width and Height specify the size of the region.

Code is a unique byte value used to identify this particular hot region for this frame. It should be given a value between hsRegion0 and 254, inclusive.

Example

```
Win.wFrame.AddHotRegion(frTL, hsRegion2, 0, 0, 2, 2);
```

Adds a two-by-two hot region nestled in the top-left corner of the frame. The region has a user code of hsRegion2.

See Also

AddHotBar AddHotHeader

Declaration

```
{$IFDEF UseHotSpots}
procedure AddHotSpot(Posn : FrameCornerType; HotChar : Char;
                     var Colors : ColorSet);
```

Purpose

Add a hot spot in a frame corner.

Description

Unlike the other methods to add hot spots, this one adds a header at the same time that it adds the spot. The header in this case is a single character string, with a value of HotChar. The header may be positioned in any of the four corners of the frame: frTL, frTR, frBL, or frBR.

The code is always hsSpot for this kind of hot spot. The four possible spots are differentiated by their frame position.

AddHotSpot extracts the fields HotSpotColor and HotSpotMono from the ColorSet passed in. It assigns these colors to the associated header.

If you add more than one hot spot per frame corner, the results derived from the frame will become ambiguous.

Example

```
Win.wFrame.AddHotSpot(frTR, #10, DefaultColorSet);
```

Adds a hot spot to the top-right corner of the frame. It will be displayed as a box character (#10) and will use default colors.

See Also

AddHotHeader

Declaration

```
{$IFDEF UseScrollBars}
procedure AddScrollBar(Posn : FrameEdgeType;
                       MinUser, MaxUser : LongInt;
                       var Colors : ColorSet);
```

Purpose

Add a standard scroll bar.

Description

Works just like AddCustomScrollBar, except that a few of the parameters take on default values. DecOffset and IncOffset both have a value of 1, so that the scroll bar fills the specified frame edge without using the frame corners. SliChar and BarChar are taken from the global typed constants DefSliderChar and DefBarChar, respectively.

Example

```
Win.wFrame.AddScrollBar(frRR, 1, 100, DefaultColorSet);
```

Adds a scroll bar to the right edge of the window with a user range of 1 to 100.

See Also

AddCustomScrollBar

Declaration

```
{$IFDEF UseShadows}
procedure AddShadow(Posn : ShadowPosType; DrawType : ShadowDrawType);
```

Purpose

Add a standard shadow.

Description

Adds either of two kinds of predefined shadows: one with the shadows to the right and bottom of the frame (Posn = shBR), and another with shadows to the left and bottom of the frame (Posn = shBL). For other kinds, use AddCustomShadow to compose the shadow as desired.

DrawType specifies whether the shadow overwrites the underlying characters or just changes their attributes. To overwrite, specify shOverWrite. In this case, the character used for the horizontal segment of the shadow is taken from the global typed constant DefShadowCharH and for the vertical segment DefShadowCharV. The width of the shadow's vertical segment is 1 character.

To change just the attributes, specify shSeeThru. In this case, the width of the shadow's vertical segment is 2 characters.

In either case, the attribute associated with the shadow is the default assigned (from the ShadowColor and ShadowMono fields of a ColorSet) when the frame was created. For a non-default color, use AddShadowColor instead.

Example

```
Win.wFrame.AddShadow(shBR, shOverWrite);
```

Adds an opaque bottom-right shadow (shadow surrounds the bottom and right edges of the window).

```
Win.wFrame.AddShadow(shBL, shSeeThru);
```

Adds a see-through, bottom-left shadow.

See Also

AddCustomShadow AddShadowColor

AddShadowColor

Declaration

```
{$IFDEF UseShadows}
procedure AddShadowColor(Posn : ShadowPosType; DrawType : ShadowDrawType;
                         AttrColor, AttrMono : Byte);
```

Purpose

Add a standard shadow with custom color.

Description

Works just like AddShadow with the exception that non-default video attributes can be specified for the shadow.

Example

```
Win.wFrame.AddShadowColor(shBR, shOverWrite, $02, $70);
```

Adds an opaque, bottom-right shadow using green on black for color monitors, reverse-video on monochrome monitors.

See Also

AddCustomShadow AddShadow

Declaration

```
procedure AddSpanHeader(FirstChar, SpanChar, LastChar : Char;
                        Dpos : Integer; Posn : FrameEdgeType);
```

Purpose

Add a header spanning the frame.

Description

Use this method to add "divider bars" to a window. This header completely spans the frame and joins into the frame's border. The span may be oriented horizontally or vertically. The span is automatically adjusted when the frame is resized.

FirstChar is the character drawn at the leftmost (for a horizontal span) or topmost (for a vertical span) edge. This character overwrites one character of the existing border and generally should be chosen to merge neatly into the border. SpanChar is the character used for the bulk of the span. LastChar merges into the frame border at the other end of the span.

Posn specifies the frame edge (frTT, frBB, frLL, or frRR) to which the span is parallel. Using frTT instead of frBB, for example, determines how the span will behave if the frame is resized. The distance between the span and the specified edge remains fixed (at DPos) during sizing.

A common use of span headers is to place a horizontal bar one row below the top edge of a frame, to separate a row of titles from columns of information. For this case, Posn should be frTT and DPos should be 2.

The colors used for the span header are the same as those for the frame, taken from the FrameColor and FrameMono fields of the ColorSet used to initialize the frame.

Span headers won't work in combination with ScrollingEntryScreen and ScrollingSelector objects (which are derived ultimately from RawWindow and hence contain Frame objects themselves). In order to produce the effect of scrolling, these objects regularly overwrite the region where span headers normally reside by copying a portion of a virtual screen to the window. You can instead use a text field (part of the entry screen) to create a span that scrolls along with the rest of the screen. The only drawback of this approach is that the scrolling span does not join smoothly with the window's border.

Examples

```
Win.wFrame.AddSpanHeader(#178, #178, #178, 0, frTT);
Win.wFrame.AddHeader(' Header ', heTC);
```

Adds a spanning header to Win's frame, replacing the top frame bar with a row of background fill characters. This is used as background for a normal header.

```
Win.wFrame.AddSpanHeader(#195, #196, #180, 2, frTT);
```

Adds a single width horizontal bar 2 rows below the top edge of the frame.

Declaration

```
procedure AddSpanHeaderColor(FirstChar, SpanChar, LastChar : Char;
                             Dpos : Integer; Posn : FrameEdgeType;
                             AttrColor, AttrMono : Byte);
```

Purpose

Add a header spanning the frame with custom color.

Description

Works just like AddSpanHeader, except that non-default video attributes can be specified for the span.

Example

```
Win.wFrame.AddSpanHeader(#195, #196, #180, 2, frTT, $03, $70);
```

Adds a single width horizontal bar 2 rows below the top edge of the frame, in green on black for color monitors and reverse video for monochrome.

AdjustFrame / ChangeHeaderAttr

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure AdjustFrame(X1, Y1, X2, Y2 : Byte);
```

Purpose

Set new coordinates and adjust related structures.

Description

New coordinates are assigned to the frame. All headers, shadows, scroll bars, and hot spots are adjusted as appropriate.

This method is normally called only by the RawWindow object. AdjustFrame performs no range checking on the specified coordinates. It can fail with an out of memory error (epFatal + ecOutOfMemory) if the heap buffers associated with span headers or shadows cannot be resized.

If you wish to adjust a frame associated with a specific window, the appropriate method to call is RawWindow.AdjustFrameCoords, which performs range checking and also adjusts other structures of the window itself.

See Also

RawWindow.AdjustFrameCoords

Declaration

```
procedure ChangeHeaderAttr(Index : Byte; Color, Mono : Byte);
```

Purpose

Change existing header colors.

Description

Index is the sequence number of the header to change. The first header added has sequence number 0, the second header 1, and so on. Typically, the index of a specific header will be stored when the header is added to the frame, using the GetLastHeaderIndex method.

If Index specifies a non-existent header, ChangeHeaderAttr generates an error (epNonFatal + ecWinBadIndex) and exits.

Color and Mono specify the new header attributes to use in color and monochrome video modes respectively.

Example

```
Win.wFrame.SetHeaderAttr($02, $70, False);
Win.wFrame.AddHeader('My Header', heTC);
...
Win.wFrame.ChangeHeaderAttr(0, $03, $70);
```

The header is originally added with a color attribute of $02 (green on black). It is later changed to $03 (cyan on black).

```
Win.wFrame.ChangeHeaderAttr(1, $03, $70);
```

Incorrect! (for the example above) The index number for the first (or only) header is zero, not one.

See Also

GetLastHeaderIndex ChangeHeaderString

Declaration

```
procedure ChangeHeaderString(Index : Byte; S : string; var Redraw : Boolean);
```

Purpose

Change existing header string and update position.

Description

Index is the sequence number of the header to change. The first header added has sequence number 0, the second header 1, and so on. Typically, the index of a specific header will be stored when the header is added to the frame, using the GetLastHeaderIndex method.

If Index specifies a non-existent header, ChangeHeaderString generates an error (epNonFatal + ecWinBadIndex) and exits.

S is the new header string. If the new string's length differs from the old and if the associated header is not of type heCustom, ChangeHeaderString recomputes the header position so that it remains centered or justified as required.

If the new string is longer than the allocated string buffer (which is the size of the longest string used for the specified header), the old buffer is disposed and a new one allocated. If this fails, ChangeHeaderString generates an error (epFatal + ecOutOfMemory).

ChangeHeaderString does not update the screen to show the new header. If the frame is already visible, this can be accomplished by calling DrawHeader, which draws just a specified header, or by calling UpdateFrame, which redraws the entire frame from scratch. The Redraw parameter is set to True upon return from ChangeHeaderString if it is necessary to call UpdateFrame instead of the faster and simpler DrawHeader. This occurs when the new header string is shorter than the old and hence would not completely overwrite it.

The RawWindow object has a somewhat higher level method (ChangeHeader) that encapsulates calls to ChangeHeaderString, DrawHeader, and UpdateFrame. You'll typically want to use it instead of ChangeHeaderString.

Example

```
var
  Redraw : Boolean;
...
Win.wFrame.AddHeader('Top Header', heTC);     {Header 0}
Win.wFrame.AddHeader('Bottom Header', heBC);  {Header 1}
...
Win.wFrame.ChangeHeaderString(1, 'New Header', Redraw);
```

Adds two headers (top and bottom), then changes the bottom header to 'New Header'. Redraw will be set to True since the new header is shorter than the old.

See Also

ChangeHeaderAttr
DrawHeader
RawWindow.ChangeHeader
UpdateFrame

ChangeHotHeader / ClearErrors

Declaration

```
{$IFDEF UseHotSpots}
procedure ChangeHotHeader(Index : Byte; Len : Byte);
```

Purpose

Change the length (and position) of a hot header.

Description

Call this routine if you have a hot spot (added by AddHotHeader) that coincides with a header and you've changed the length of the header by calling ChangeHeaderString. ChangeHotHeader will adjust the length and position of the hot spot to match the header again.

Index is the sequential number of the hot spot (which is likely to be different from the index of the associated header). You can count this yourself (starting with 0 for the first hot spot) or use the GetLastHotIndex function to store the index when the hot spot is added.

Len is the length of the associated header.

Example

```
Win.wFrame.AddHotHeader(heTC, 3, Win.Width);
...
Win.wFrame.ChangeHotHeader(0, 20);
```

Adds a hot header for the total width of the window and later changes that width to 20 columns.

See Also

AddHotHeader GetLastHotIndex

Declaration

```
procedure ClearErrors;
```

Purpose

Clear all pending errors.

Description

After calling ClearErrors, the next call to GetLastError will always return zero, even if several different errors were pending. (For now, only one error at a time is possible for the Frame object, but that could change in the future.)

ClearErrors is actually implemented in the AbstractFrame object, from which Frame is derived.

Example

```
Win.wFrame.ClearErrors;
```

Clears any pending errors associated with wFrame.

See Also

GetLastError

Declaration

```
procedure Coordinates(var X1, Y1, X2, Y2 : Byte);
```

Purpose

Return the coordinates of the frame.

Description

Coordinates are returned in absolute terms (the upper left corner of the screen is X=1, Y=1). The coordinates include the border of the frame, if any, but do not include any headers or shadows that fall outside of the border.

Example

```
var
  Xlo, Ylo, Xhi, Yhi : Byte;
...
Win.wFrame.Coordinates(Xlo, Ylo, Xhi, Yhi);
WriteLn('Upper left corner:  ', Xlo, ',', Ylo);
WriteLn('Lower right corner: ', Xhi, ',', Yhi);
```

Displays the screen coordinates of Win's frame.

See Also

RawWindow.Extent WithinFrameCoords

Declaration

```
{$IFDEF UseScrollBars}
function CurUserValue(Posn : FrameEdgeType) : LongInt;
```

Purpose

Return the current user value for specified scroll bar.

Description

If a user value has been assigned to the scroll bar (typically by drawing the slider), it is returned by this function. If no value has been assigned, CurUserValue returns 0.

Example

```
var
  SliderPos : LongInt;
...
Win.wFrame.AddScrollBar(frRR, 1, 100, DefaultColorSet);
Win.Draw;
Win.DrawSlider(frRR, 50);
...
SliderPos := Win.wFrame.CurUserValue(frRR);
```

SliderPos gets 50, the current value of the right-edge scroll bar's slider.

See Also

RawWindow.DrawSlider SetSliderValue

DisableHeader / Done

Declaration

```
procedure DisableHeader(Index : Byte; On : Boolean);
```

Purpose

Disable or enable specified header.

Description

Disabled headers are not drawn by Frame.Draw, UpdateFrame, or DrawHeader. Such behavior is useful when two headers share the same location and one or the other appears depending on the circumstances. For example, consider a window that displays a filename at the top center of the frame. When the window is to be moved or resized, you might want to replace the normal header with one providing move instructions. In this case, you could temporarily disable the normal filename header, enable the instruction header, and update the frame. After the move was complete, you would reverse the state and update the frame again.

Index is the sequence number of the header to change. The first header added has sequence number 0, the second header 1, and so on. Typically, the index of a specific header will be stored when the header is added to the frame, using the GetLastHeaderIndex method.

On is True to disable a header, False to enable it. It is not an error to disable an already disabled header. All headers are enabled by default.

If Index specifies a non-existent header, DrawHeader generates an error (epNonFatal+ecWinBadIndex) and exits.

Example

```
...
Win.wFrame.AddHeader('Main Header', heTC);
MainIndex := Win.wFrame.GetLastHeaderIndex;
Win.wFrame.AddHeader('Alternate Header', heTC);
AltIndex := Win.wFrame.GetLastHeaderIndex;
Win.wFrame.DisableHeader(AltIndex, True);
Win.Draw;
...
Win.wFrame.DisableHeader(AltIndex, False);
Win.wFrame.DisableHeader(MainIndex, True);
Win.wFrame.UpdateFrame;
```

Adds two headers, both to the top center position of the frame, and saves the index number of each. Disables the alternate header and draws the window. Later enables the alternate header, disables the main one, and updates the frame to show the alternate header.

See Also

GetLastHeaderIndex UpdateFrame

Declaration

```
destructor Done; virtual;
```

Purpose

Destroy a frame.

Description

Deallocates all the subsidiary structures associated with the frame, including the headers, shadows, scroll bars, and hot spots.

Example

```
Win.wFrame.Done;
```

Deallocates all heap memory used the Win's main frame.

Declaration
```
procedure Draw; virtual;
```

Purpose
Draw the frame, scrollbars, headers, and shadows.

Description
With the exception of shadows, Frame.Draw does not save the contents of the underlying screen before writing. Screen save and restore are handled by the RawWindow object, which manages the associated frames for you. Each shadow rectangle does handle its own save and restore, however. If any header falls outside of the frame, it is the programmer's responsibility to save and restore the region it overwrites; even the RawWindow object does not attempt to handle this situation.

Disabled headers are not drawn.

If a scrollbar's current user value has been initialized (by calling SetSliderValue or DrawSlider previously), the slider is drawn during the call to Frame.Draw. Otherwise, only the background of the bar is drawn.

Example
```
Win.wFrame.Draw;
```
Draws all frames, headers, sliders and hotspots for Win.

See Also
UpdateFrame

Declaration
```
procedure DrawHeader(Index : Byte);
```

Purpose
Draw specified header.

Description
This method is useful when a header is changed after the frame has been drawn. If the new header completely covers the old, DrawHeader will correctly update it. Otherwise, UpdateFrame must be used to draw the frame from scratch.

Index is the sequence number of the header to change. The first header added has sequence number 0, the second header 1, and so on. Typically, the index of a specific header will be stored when the header is added to the frame, using the GetLastHeaderIndex method.

If Index specifies a non-existent header, DrawHeader generates an error (epNonFatal + ecWinBadIndex) and exits. If the specified header has been disabled, DrawHeader exits without drawing it.

If the header starts within the frame, DrawHeader automatically clips it as needed to stay within the frame.

Example
```
Win.wFrame.DrawHeader(0);
```
Draws Win's first header.

See Also
ChangeHeaderAttr ChangeHeaderString

Error / fCopy

Declaration

```
procedure Error(Code : Word); virtual;
```

Purpose

Report that an error occurred.

Description

The Frame object's methods call this routine whenever they detect an error. Error simply stores the error code in a variable internal to the object, where it can be accessed by the GetLastError method. You typically won't make calls to the Error method in your own code.

You may want to override the Error method in an object derived from Frame in order to handle errors differently.

See Also

GetLastError

Declaration

```
constructor fCopy(var F : Frame);
```

Purpose

Initialize a frame from another, making a unique copy of all dynamic structures.

Description

This method clones one Frame object (F) to initialize another (Self). Where the frame contains dynamic objects such as headers, shadows, etc., additional copies of each one are allocated and linked into the new object. In this way, the new frame can be modified without affecting the old.

fCopy is useful when considered along with the RawWindow object, which includes two separate frames. One frame is displayed when the window is current and the other when the window is not current (provided that the RawWindow's wAltFrame option is set). For example, you might want the non-current frame to use a different video attribute for its headers and a different set of characters for its border. Typically, however, the two frames will be more alike than different, and a good approach to initializing the alternate frame is to copy the main frame (using fCopy), then modify it as desired.

fCopy will generate an error (epFatal+ecOutOfMemory) if there is insufficient heap space available to allocate the new frame structures. Because fCopy is a constructor, the error code is returned in the global typed constant InitStatus.

Example

```
Win.aFrame.fCopy(Win.wFrame);
```

Clones the window's main frame (wFrame) onto its alternate frame (aFrame).

See Also

Init RawWindow.wAltFrame

Declaration

```
function GetLastError : Word; virtual;
```

Purpose

Return and clear the last error code.

Description

GetLastError returns zero if no error is pending. Otherwise it returns the combination of an error prefix (a multiple of 10000) and an error code (1..9999). Error codes are listed in Appendix D. Specific errors that can occur as a result of calling particular methods are mentioned in the documentation for each method.

For the Frame object, it is most important to call GetLastError after each call (or sequence of calls) to an Add method.

Like Turbo Pascal's IoResult function, GetLastError clears the internal error variable after each call. Immediately calling GetLastError again will return zero.

Example

```
Win.wFrame.AddCustomScrollBar(frRR, 0, 9, 1, 1, #7, #179, DefaultColorSet);
Win.wFrame.AddShadow(shBR, shOverWrite);
Win.wFrame.AddHeader('My Header', heTC);
Status := Win.wFrame.GetLastError;
if Status <> 0 then begin
  WriteLn('Failed to add window features, Status = ', Status);
  Halt;
end;
```

Checks for errors after adding three window features.

See Also

ClearErrors

Declaration

```
function GetLastHeaderIndex : Byte;
```

Purpose

Return index of last header added.

Description

This routine is useful when you are adding a header and you expect to modify the header's string or color after the window is initialized. GetLastHeaderIndex returns the index of the last header added (with AddHeader, AddSpanHeader, etc.). If you save this index in a variable, you can later pass it as a parameter to ChangeHeaderString, ChangeHeaderAttr, or DrawHeader. It's better to get the header index by calling GetLastHeaderIndex than to compute the index manually; the manual approach too easily becomes obsolete when the program is modified.

GetLastHeaderIndex returns 255 if there are no headers on the frame. The first header added has an index of zero.

Example

```
Win.wFrame.AddHeader('Top Header', heTC);
Win.wFrame.AddHeader('Bottom Header', heBC);
BotIndex := Win.wFrame.GetLastHeaderIndex;
```

BotIndex gets the index of the last header added, in this case 1.

See Also

ChangeHeaderString GetLastHotIndex

GetLastHotIndex / Init

Declaration

```
{$IFDEF UseHotSpots}
function GetLastHotIndex : Byte;
```

Purpose

Return index of last hot region.

Description

Performs the same service for hot spots that GetLastHeaderIndex does for headers. You may wish to modify hot spots after adding them in order to keep them consistent with an associated header.

Note that headers and hot spots have independent sequence numbers. There is no connection between the two.

Example

```
HotIndex := Win.wFrame.GetLastHotIndex;
```

Returns the index of the last hot region (zero-based).

See Also

GetLastHeaderIndex

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; FA : FrameArray; var Colors : ColorSet);
```

Purpose

Initialize frame coordinates and all fields.

Description

You won't need to call this method directly unless you're implementing new data types. The RawWindow object and its derivatives call Frame.Init with the appropriate parameters when a window is initialized. The following description is provided for the sake of completeness.

X1, Y1, X2, and Y2 are the absolute coordinates of the frame's corners, including the border if any. FA specifies the border characters. The following predefined constants are available to pass for this parameter:

```
DefWindowFrame    double bar horizontal, single bar vertical
DblWindowFrame    double bar all the way around
SglWindowFrame    single bar all the way around
NoWindowFrame     no border
```

No border is indicated by passing the character NoFrameChar (#255) in the appropriate positions of the frame array. Note that it's possible to have border on some sides of the frame and not others; just fill in the frame array appropriately.

The following fields in the ColorSet are actually used by the Frame object: FrameColor, FrameMono, HeaderColor, HeaderMono, ShadowColor, and ShadowMono. FrameColor and FrameMono are the attributes used to draw the frame's border. HeaderColor and HeaderMono are used for each header (unless specified otherwise when the header is added). ShadowColor and ShadowMono apply to shadows unless specified otherwise.

Init also sets the frame's sizing and clip limits to default values. If the window has any visible border characters, the minimum sizing limits for the frame are two characters wide and two high; unbordered frames can shrink to one by one. The clip limits are set to allow writing to the entire screen in the current mode, X=1..ScreenWidth and Y=1..ScreenHeight. See SetSizeLimits and SetClipLimits for more information.

Frame.Init currently has no way to fail. Even so, it's a good idea to check that the constructor succeeded, and to refer to the InitStatus variable in case of error.

See Also

SetClipLimits SetSizeLimits

Declaration

```
function IsFramed : Boolean;
```

Purpose

Return True if any of the FrameChars are not NoFrameChar.

Description

Frames may be bordered or not. IsFramed returns True if any portion of the frame's border will be visible, that is, if it contains any character besides NoFrameChar.

Example

```
if Win.wFrame.IsFramed then
  Win.wFrame.SetFrameType(DblWindowFrame);
```

If Win's frame currently has a border, change it to a double-line border style.

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load from stream.

Description

You won't need to call this method directly unless you're implementing new data types. The RawWindow object and its derivatives call Frame.Load when a window is loaded. The following description is provided for the sake of completeness.

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Frame.Store. Load initializes the frame coordinates and adds any headers, shadows, scroll bars, and hot spots that were part of the frame when it was stored.

If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.

The stream registration procedure for a Frame is FrameStream.

See Also

RawWindow.Load Store

SetClipLimits / SetFrameAttr / SetFrameType

Declaration
```
procedure SetClipLimits(MinXL, MinYL, MaxXH, MaxYH : Byte);
```

Purpose
Set limits for drawing frame (shadows get clipped).

Description
The clip limits define a rectangular screen region beyond which shadows will not be drawn. If you try the DESKPOP demo program, you'll notice that shadows never overwrite the status line at the bottom of the screen and that windows can be resized to fill the width of the screen, making the shadows effectively disappear. A frame's shadows are designed to be optional; they appear in a region controlled by the clip limits.

By default, the clip limits cover the entire screen. Use SetClipLimits to restrict the region.

Example
```
Win.wFrame.SetClipLimits(1, 1, 60, 20);
```
Sets clip limits to a rectangle with upper left corner at 1,1 and lower right corner at 60,20.

See Also
SetSizeLimits

Declaration
```
procedure SetFrameAttr(Color, Mono : Byte); virtual;
```

Purpose
Set attributes for border characters.

Description
Use this method to change the video attributes for the frame's border. Color is the attribute used for a color video mode, and Mono for monochrome.

Example
```
Win.wFrame.SetFrameAttr($02, $70);
```
Sets Win's border attribute to green on black for color monitors, reverse video for monochrome monitors.

See Also
Init SetHeaderAttr

Declaration
```
procedure SetFrameType(FA : FrameArray);
```

Purpose
Change the border type.

Description
Use this method to change the line drawing characters specified for a frame's border. FA is an array of eight characters specifying the four corners and four edges of the frame. See FrameArray in the Types section above. Also see the several standard values that may be passed for FA in the Constants section above.

Example
```
Win.wFrame.SetFrameType(DblWindowFrame);
```
Sets Win's main frame to double-line frame characters.

Declaration

```
procedure SetHeaderAttr(Color, Mono : Byte; ChangeAll : Boolean);
```

Purpose

Set attributes for header strings.

Description

SetHeaderAttr's first effect is to change the default attribute used when further headers are added to the frame.

If the ChangeAll parameter is True, SetHeaderAttr has a second effect: it changes the attributes of all headers added *so far*. Be careful using this option; even headers that default to attributes other than the normal header attribute (span headers, for example), are changed.

Color and Mono are the video attributes used for color and monochrome video modes, respectively.

Example

```
Win.wFrame.SetHeaderAttr($03, $70, False);
```

Sets future headers to use cyan on black for color monitors, reverse video for monochrome monitors.

Declaration

```
{$IFDEF UseShadows}
procedure SetShadowAttr(Color, Mono : Byte; ChangeAll : Boolean);
```

Purpose

Set attributes for shadow characters.

Description

SetShadowAttr's first effect is to change the default attribute used when further shadows are added to the frame.

If the ChangeAll parameter is True, SetShadowAttr has a second effect: it changes the attributes of all shadows added *so far*.

Color and Mono are the video attributes used for color and monochrome video modes, respectively.

Example

```
Win.wFrame.SetShadowAttr($0E, $07, True);
```

Sets Win's shadow attribute to yellow on black for color monitors, white on black for monochrome monitors and applies this change to all existing shadows.

SetSizeLimits / SetSliderValue

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure SetSizeLimits(MinW, MinH, MaxW, MaxH : Byte);
```

Purpose

Set limits for sizing of frame.

Description

Limits provided here specify the range over which the AdjustFrame method can operate. Setting these limits is useful to assure that embellishments added to the frame--headers and scroll bars especially--have enough room to perform a useful function. The RawWindow object enforces the limits set here; as mentioned previously, AdjustFrame does not perform range checking itself.

MinW and MinH are the minimum width and height of the frame; these are generally the most critical to specify. MaxW and MaxH are the maximums.

Example

```
Win.wFrame.SetSizeLimits(10, 10, 60, 20);
```

Sets frame size limits for Win.wFrame. The smallest size is 10 columns by 10 rows; the largest size is 60 columns by 20 rows.

See Also

AdjustFrame SetClipLimits

Declaration

```
{$IFDEF UseScrollBars}
procedure SetSliderValue(Posn : FrameEdgeType; UserVal : LongInt);
```

Purpose

Change the user value without drawing it.

Description

Posn is a frame edge containing a scroll bar: frTT, frBB, frLL, or frRR. If there is no scroll bar on that edge, SetSliderValue exits immediately. Otherwise, it updates the internal variables storing the current user value and physical slider position. It does nothing to update the screen; call the RawWindow.DrawSlider method for that.

This method is useful when updating a slider on a frame that is not currently on-screen.

Example

```
Win.wFrame.SetSliderValue(frRR, CurRow);
```

Sets the slider position to a value corresponding to CurRow, a hypothetical current row in a pick list.

See Also

CurUserValue RawWindow.DrawSlider

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store frame to stream.

Description

You won't need to call this method directly unless you're implementing new data types. The RawWindow object and its derivatives call Frame.Store when a window is stored. The following description is provided for the sake of completeness.

S is a properly initialized stream object (a stream file opened for writing, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Store writes the frame coordinates and calls the store methods for any headers, shadows, scroll bars, and hot spots that have been added to the frame.

If Store fails, the stream's GetStatus method will return a non-zero value.

The stream registration procedure for a Frame is FrameStream.

See Also

Load RawWindow.Store

Declaration

```
procedure UpdateFrame; virtual;
```

Purpose

Draw the frame, scrollbars, and headers (no shadows).

Description

This method draws the frame's border, its headers, and its scroll bars, in that order. It does not draw the shadows. UpdateFrame is intended for situations where the frame has already been drawn once, is visible on screen, and has undergone a significant change such as the modification of a header. UpdateFrame will show this change.

If the frame has not yet been drawn, call Frame.Draw instead.

Example

```
Win.wFrame.UpdateFrame;
```

Draws Win's frame, scrollbars, and headers.

See Also

Draw DrawHeader

WithinFrameCoords

Declaration

```
procedure WithinFrameCoords(var X1, Y1, X2, Y2 : Byte);
```

Purpose

Return the coordinates of region within, not including, the border.

Description

For frames with borders, this method returns different results than the Coordinates method, which includes the border characters. WithinFrameCoords is used by the RawWindow object to determine the region to clear for the ClearContents method, among other actions.

Example

```
var
  InX1, InY1, InX2, InY2, I : Byte;
...
Win.wFrame.WithinFrameCoords(InX1, InY1, InX2, InY2);
for I := InY1 to InY2 do
  FastText(CharStr(#9, InX2-InX1+1), I, InX1);
```

Fills the area within the border with background characters.

See Also

Coordinates

C. OPWINDOW: Window Objects

OPWINDOW picks up where OPFRAME leaves off, providing the next level of sophistication in the window hierarchy. (Initially OPFRAME and OPWINDOW were combined within one unit, but this strained the limits of the Turbo Pascal compiler, so they were split.) The following chart shows the objects implemented by OPWINDOW.

```
[Root]
  LoadableColorSet
  PackedWindow
  [PointerStack]
    WindowStack
  VirtScreen
  AbstractWindow
    RawWindow
      StackWindow
        CommandWindow
```

Of these, the one you're most likely to declare as a variable is the StackWindow. For an overview of the others, see the introduction to this chapter. We'll save further discussion of LoadableColorSet, PackedWindow, and VirtScreen for later in this section.

There isn't much concrete to say about AbstractWindow, so we'll just mention it briefly here. AbstractWindow describes only the most basic properties of a window. We provide it in case we (or you) need to make fundamental extensions to the way windows currently work in **Object Professional**. Possible examples include graphics mode or multi-pane windows (like Turbo Debugger's CPU window). It's better to derive such extensions from an ancestor also shared by RawWindow and the other **Object Professional** window objects. In that way, you can still create a linked list (or another unifying structure) of all these window-related objects.

If you find yourself straining to implement a particular visual or logical effect with a type derived from a RawWindow, you should spend some time studying the definition of AbstractWindow in OPWINDOW.PAS. You may be able to derive a new type that matches the specifications of AbstractWindow while adding the functionality that you need.

Of the methods provided by AbstractWindow, only three are not overridden by RawWindow. These three--ClearErrors, Height, and Width--are described in the RawWindow section just to make them easier to find.

Window Coordinates

At its simplest, a window is a rectangle on the screen, described by two sets of (X,Y) coordinates, one for the upper left corner of the window and the other for the lower right. However, by the time a window is fully implemented (in RawWindow and its derivatives), several sets of coordinates are involved. It's important to understand the distinctions between them.

We'll start off by listing the various critical dimensions and the methods that control them. Then we'll describe each one in detail.

- active window rectangle (Init, InitCustom)
- framed region rectangle (AdjustFrameCoords)
- minimum and maximum active width and height (SetSizeLimits)
- minimum and maximum frame width and height (SetFrameLimits)
- frame limiting rectangle (SetPosLimits)

You're probably most familiar with the active window rectangle. Its coordinates are equivalent to the ones used for the Window procedure from the CRT or OPCRT unit. These coordinates specify the region where Write and WriteLn work and where scrolling occurs. The statement GotoXY(1, 1) moves the cursor to the upper left corner of the active window rectangle. Window-relative fast-writing methods such as wFastWrite use the active window rectangle as their reference; they also clip text that would extend beyond this window.

The framed region rectangle may be identical to the active window, but it is not required to be so. In fact, for windows with a border, the framed region surrounds the active window by one character position on all sides. RawWindow also provides a method (AdjustFrameCoords) that lets you increase the size of the framed region arbitrarily. This method is useful when you want to write fixed titles outside of the window's scrolling region but within the frame's border. You can't use WriteLn statements to write these titles because they are outside of the active window, but RawWindow provides a full range of frame-relative fast-writing methods for this purpose.

When RawWindow and its derivatives save the contents of the screen region covered by a window, the coordinates used are those of the framed region rectangle. When these objects save the contents of the window itself, the coordinates are just inside of the border (if any) of the framed region.

You must always specify the *active window* coordinates when calling a window's constructor (usually Init or InitCustom). If the window has a border, remember to reserve at least one position on each edge for the border. For example, it is an error to specify the coordinates (1, 1, 80, 25) for a bordered window on a normal screen. If you've used Turbo Professional's window system before, you'll have a mental shift to make; **Object Professional**'s consistent use of the *active* coordinates to initialize each window is a change from Turbo Professional.

The remaining three quantities control a window's behavior when requests are made to resize or reposition it. If your windows' coordinates aren't modified at runtime, you probably won't need to consider the following.

The window manager checks the minimum and maximum active width and height before resizing a window. These limits default to 1 by 1 and full-screen, respectively. You can modify these limits by calling the SetSizeLimits method when the window contents warrant it. For example, the window might contain a pick list with minimum allowable width of 10 and maximum of 30 columns.

The window manager also checks the minimum and maximum frame width and height before resizing the frames associated with each window. For a bordered window, the default minimum is 2 by 2; for non-bordered, 1 by 1. These limits are usually associated with particular headers or scroll bars. Whenever a scroll bar is added to a frame the minimum will be raised if required. In this way the frame can never be shrunk below the size needed to maintain a valid scroll bar. You can explicitly modify the frame size limits by calling the SetFrameLimits method.

Finally, the frame limiting rectangle controls where a window may be moved on the screen. The outer edge of the frame may not move outside the frame limiting rectangle. The same rectangle also serves as a clipping boundary for shadows and headers. The frame limiting rectangle is useful when there are menus or status lines on-screen that no window may obscure (DESKPOP uses this technique). You can adjust the frame limiting rectangle by calling SetPosLimits.

Conditional Defines

The objects in OPWINDOW, like those in OPFRAME, are affected by the following conditional defines:

```
{$DEFINE UseMouse}
{$DEFINE UseScrollBars}
{$DEFINE UseHotSpots}
{$DEFINE UseShadows}
{$DEFINE UseAdjustableWindows}
```

Edit OPDEFINE.INC and deactivate those defines you don't need. Doing so will reduce the executable size of your program and allow the compiler to handle more lines of your source code.

See the Conditional Defines section in Section B.2 (OPFRAME) for more information.

Windows and Streams

All of the objects in OPWINDOW can be read from and written to streams. The following procedures are interfaced by OPWINDOW to perform type registration duties. See Chapter 8 for background, and the Load and Store methods of each object for more information.

```
procedure RawWindowStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing raw windows}
procedure ExplodingWindowStream(SPtr : IdStreamPtr);
  {-Register procedures used for exploding windows}
procedure StackWindowStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing stacked windows}
procedure CommandWindowStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing command windows}
procedure WindowStackStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing window stacks}
procedure VirtScreenStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing virtual screens}
procedure LoadableColorSetStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing loadable color sets}
procedure PackedWindowStream(SPtr : IdStreamPtr);
  {-Register all types needed for streams containing packed windows}
```

1. RawWindow

RawWindow implements most of the functions you would expect from a text window. It provides the buffers needed to save the screen region covered by and contained within the window, the logic needed to display the window's frame and set its active coordinates, and methods for clearing, writing to, and scrolling within the window.

What RawWindow does not offer is the management of relationships between multiple windows. That capability is added by the StackWindow and WindowStack objects. The only such distinction made by RawWindow is whether a given window is *current* (the window drawn most recently), *active* (on screen, but not necessarily current), or *inactive* (not on screen at all).

Given this limitation, a RawWindow is most useful in an environment where the windows are tiled (non-overlapping) and where control is transferred from window to window by drawing each one in turn. Since the StackWindow object has only a slight additional overhead, it is the window type of choice for most applications. Nevertheless, the StackWindow derives many of its properties from RawWindow, so the capabilities of RawWindow are worth investigating.

The following example shows how to use some features of the RawWindow, including how to switch between two non-overlapping windows.

```
program RawWindowExample;  {EXRAWWIN.PAS}
uses
  OpRoot, OpString, OpCrt, OpFrame, OpWindow;
var
  W1, W2 : RawWindow;

procedure Pause;
  {-Wait for a keypress}
var
  KW : Word;
begin
  KW := ReadKeyWord;
end;

begin
  ClrScr;

  {Initialize first window}
  W1.InitCustom(10, 5, 40, 20, DefaultColorSet, wClear+wBordered+wAltFrame);
  W1.EnableExplosions(20);
  W1.wFrame.AddHeader(' First Window ', heTL);
  W1.aFrame.AddHeader(' inactive now ', heTL);
  W1.aFrame.SetFrameType(SglWindowFrame);

  {Initialize second, non-overlapping window}
  W2.InitCustom(45, 10, 75, 15, DefaultColorSet, wClear+wBordered);
  W2.wFrame.AddHeader(' Second Window ', heBC);
  W2.wFrame.AddShadow(shBR, shOverWrite);

  {Draw first window, write to it, change to block cursor}
  W1.Draw;
  WriteLn('In the first window');
  WriteLn('Change to block cursor');
  W1.SetCursor(cuBlock);
  Pause;

  {Draw second window, write to it}
  W2.Draw;
```

```
    Writeln('In the second window');
    W2.wFastText('Text can be clipped when it is too long to fit', W2.Height, 1);
    W2.wGotoXY(W2.Width, W2.Height);
    Pause;

    {Switch back to first window}
    W1.Draw;
    Pause;

    {Erase both windows, starting with the second}
    W2.Erase;
    Pause;
    W1.Erase;

    {Dispose of windows}
    W2.Done;
    W1.Done;
  end.
```

The example declares two static instances of RawWindows, W1 and W2. W1 is initialized as a bordered window by using the wBordered option. Because the wAltFrame option is specified, W1 manages two separate instances of a Frame object: It displays wFrame (a field within the RawWindow object) when W1 is the current window; and instead displays aFrame (another field) when W1 is active but not current. To make the difference obvious in the example, we use a different header and border style for the alternate frame. W1 is also exploding, a special effect achieved by calling EnableExplosions. Explosions are a special case of the more general capability to call a user-defined routine whenever a window is opened or closed.

W2 is also initialized as a bordered window, but it doesn't take advantage of the alternate frame option. W2's frame has a shadow. (These frame-related features–borders, headers, shadows–are the same ones described in the previous section.)

The example program then displays the window W1 by calling its Draw method. The Draw method saves the underlying screen and cursor state, calls the user-defined open method (which creates the exploding effect), clears the window (because the wClear flag was set), sets the OPCRT window coordinates and default text attribute, and calls W1.wFrame's Draw method to display the border and header. RawWindow.Draw also calls a virtual method named UpdateContents, which does nothing at this level but plays a very important role for derived types. The UpdateContents method provides an opportunity for the derived object to fill in the window's contents (with a pick list or editor text, for example).

After calling Draw, it's possible to write text within the window by calling non-object-oriented procedures. Write and WriteLn work fine; they use the video attribute associated with the window, wrap text if required, and scroll the window when it is full. GotoXY also works with coordinates relative to the active portion of the window.

Next we draw the window W2 and write some text within it. If you run the example program, you'll note that W1's frame changes to the alternate version when W2 is displayed. The call to wFastText illustrates important differences between WriteLn statements and the object-oriented writing methods: the OOP versions clip text that would cause WriteLn to wrap and scroll; and the OOP versions can write anywhere within the window regardless of the cursor position.

A second call to W1.Draw makes W1 the current window again. During this call, Draw detects that W1 is already active and doesn't save the underlying screen again or call the user-defined open procedure.

Finally, both windows are erased and deallocated. Note that W2, which is not the current window at the time, is erased first. RawWindow.Erase would perform this action even if W2 and W1 overlapped, but the result would be incorrect. Erasing the current window, W1, would probably lead to unexpected consequences in a real program since RawWindow doesn't know which window to make current next.

Constants

```
wBordered          = $00000001;  {Make space for window border?}
wClear             = $00000002;  {Clear the window when it is first displayed?}
wSaveContents      = $00000004;  {Save the contents of the window when erased?}
wUserContents      = $00000008;  {User will restore contents on request?}
wSoundEffects      = $00000010;  {Use sound effects during explosions?}
wAltFrame          = $00000020;  {Use alternate frame type when non-current?}
wSetMouse          = $00000040;  {Set mouse window, cursor on entry to window?}
wFullMouseWindow   = $00000080;  {Force to full mouse window on entry?}
wResizeable        = $00000100;  {Can be resized and zoomed?}
wAllMouseEvents    = $00000200;  {Mouse events frOutsideFrame and frInsideFrame
                                  cause an exit?}
wStoreContents     = $00000400;  {Store window contents in streams?}
```

Options that affect the behavior of a RawWindow. Bit-mapped combinations of these options are passed to InitCustom, wOptionsOn, wOptionsOff, and other methods.

wBordered selects a bordered window. As discussed in the section entitled "Window Coordinates" above, this option increases the dimensions of the window's frame by one character on all sides; the active region of the window remains unchanged.

When wClear is set, the window will be cleared automatically the first time it is displayed. It will also be cleared when drawn subsequently if the wSaveContents flag is not set.

wSaveContents causes the window's constructor to allocate an additional screen buffer, large enough to hold the entire window out to (but not including) the border. The window's Erase method will then save the window's contents to this buffer just before restoring the underlying screen. Subsequent calls to RawWindow.Draw will restore the window's contents by using this buffer.

wUserContents indicates that the object is capable of restoring its own contents. This typically occurs when an object derived from RawWindow overrides the UpdateContents method. For example, a Memo window restores the contents from the memo text buffer and a PickList restores them from the list of items. **Object Professional**'s types set this flag for themselves when appropriate. When wUserContents is set, it is usually a waste of heap space to set wSaveContents as well.

wSoundEffects enables sound effects during window explosions. This flag is relevant only when EnableExplosions is called.

wAltFrame specifies that the window will manage two independent frames: one (referred to as wFrame) when the window is current, and another (aFrame) when the window is active but not current.

wSetMouse indicates that the mouse window, cursor position, and cursor visibility should be updated when the window is drawn. In this case, the mouse will be restricted to movements within the window (including the frame border) and the mouse cursor will regain the state it had when the window was last current. wFullMouseWindow overrules wSetMouse: it forces the mouse window to full screen when the window is drawn, but does not change the mouse cursor position or visibility.

wResizeable determines whether the window can be resized and zoomed.

wAllMouseEvents affects only the descendants of CommandWindow. When set, pressing any combination of mouse buttons will cause the Process method of the CommandWindow to exit immediately, no matter where the mouse cursor happens to be at the time. When clear, the Process method will ignore all mouse clicks outside of the window.

wStoreContents specifies that the contents of a window should be written to a stream along with the specification for the window. This is possible only if the wSaveContents flag is also set.

```
DefWindowOptions  : LongInt = wClear+wSaveContents+wResizeable+wStoreContents;
```

Default window options for the Init constructor. Note that windows have no border by default.

```
DefCursor         : CursorType = cuNormal; {Cursor type}
```

Default cursor shape for a newly initialized window. See the definition of CursorType in the OPCRT unit. After a window has been initialized, change its cursor shape using the SetCursor method.

```
DefBackChar       : Char = ' ';  {Background character}
```

Default background character (used when clearing a window or scrolling it). Change the background character using the SetBackChar method.

```
CurrentWindow     : RawWindowPtr = nil;
```

A pointer to the current window. If no window is visible, the pointer is nil. Whenever a window is drawn, its address is stored in CurrentWindow. When the current window is erased (and it is a RawWindow), the value nil is stored in CurrentWindow. If the current window is erased and it is a StackWindow (or descendant), the next window on the stack becomes the current window.

Types

```
AbstractWindowPtr = ^AbstractWindow;
AbstractWindow    =
  object(Root)
    ...
  end;
```

Data types defining the most abstract window type.

```
RawWindowPtr      = ^RawWindow;
RawWindowProc     = procedure(WPtr : RawWindowPtr);
RawWindow         =              {A window without stacking support}
  object(AbstractWindow)
    ...
    wFrame        : Frame;
    aFrame        : Frame;
    ...
  end;
```

RawWindowProc is a procedure type used for user-defined window open and close routines. ExplodeOpenProc and ExplodeCloseProc, which generate exploding window special effects, are examples of routines compatible with this type. A RawWindowProc is passed a pointer to a RawWindow (or its descendant) so that it can access the data fields and methods of the window while performing the open or close operation.

wFrame and aFrame are the two frames associated with each RawWindow. They are shown explicitly here because code that customizes each frame often refers to them directly. This practice is not the best example of pure object-oriented programming, but it is efficient and convenient and we've fallen into a habit of doing it. RawWindow also provides methods to access these fields; see MainFramePtr and AltFramePtr.

RawWindow also contains two instances of a ScreenRect. These fields are used to store the screen region that the window covers, and optionally the contents of the window itself.

ActiveFramePtr

Declaration

```
function ActiveFramePtr : AbstractFramePtr; virtual;
```

Purpose

Return a pointer to frame the window is currently displaying, nil if none.

Description

If the window is current, ActiveFramePtr returns the address of the wFrame field of the instance. If the window is on-screen but not current and the wAltFrame flag has been set, ActiveFramePtr returns the address of aFrame. If the wAltFrame flag is not set, ActiveFramePtr returns the address of wFrame instead. If the window has not been drawn, ActiveFramePtr returns nil.

In order to provide maximum flexibility, ActiveFramePtr returns a pointer of type AbstractFramePtr. In order to treat the result as a pointer to a Frame object, it must be typecast as shown in the example following.

Example

```
var
  Win : RawWindow;
...
  Win.Draw;
  FramePtr(Win.ActiveFramePtr)^.ChangeHeaderAttr(0, $1B, $07);
```

Changes the video attribute of the first header on the main frame of window Win. The FramePtr typ ast is required to access the methods of the Frame object.

See Also

AltFramePtr MainFramePtr

Declaration

```
procedure AdjustFrameCoords(X1, Y1, X2, Y2 : Byte);
```

Purpose

Change frame coordinates of an inactive window. Completely clears existing frame definition.

Description

Use this routine when you want to create gap between the active portion of the window and its border. This space may be used to display additional titles that don't scroll when the active window does. The Calendar window in the DESKPOP demo program is a good example: the days of the month are managed by a pick list in the active portion of the window; the month name and days of the week are fixed titles drawn in the gap between the active window and the border.

The coordinates of both wFrame and aFrame are set to those specified by the parameters to AdjustFrameCoords. If the wBordered flag is set, the window's border will coincide with the specified coordinates. The ScreenRect that buffers the screen region covered by the window is reallocated to match the new coordinates. If the wSaveContents flag is set, the window's contents buffer is reallocated to match the region just inside of the new border (if any).

The active coordinates of the window are not changed by AdjustFrameCoords.

Any headers, shadows, scroll bars, or hot spots associated with either wFrame or aFrame are lost after this call. The colors and border characters associated with each frame are maintained, however. It's best to call AdjustFrameCoords immediately after initializing a window.

The window must be inactive (i.e., initialized but not on screen) or else AdjustFrameCoords will exit immediately with error epNonFatal+ecWinNotActive. If the specified coordinates are invalid (less than one or greater than the screen size), the error epFatal+ecWinCoordsBad is generated (the window is then unusable). If insufficient memory is available to reallocate screen buffers, epFatal+ecOutOfMemory is returned.

Examples

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.AdjustFrameCoords(29, 3, 51, 17);
Win.wFrame.AddHeader('Standard Header', heTC);
Win.Draw;
Win.fFastCenter('Fixed Top Heading', 1, TextAttr);
Win.fFastCenter('Fixed Bottom Hdr', 13, TextAttr);
{...Use Window...}
Win.Done;
```

The top and bottom border bars are moved one row out (without changing the dimensions of the active window). The frame relative writing methods (fFastXxx) are used to write fixed headings in this space.

```
Win.Init(30, 5, 50, 15);
Win.wFrame.AddHeader('Standard Header', heTC);
Win.wFrame.AddShadow(shBR, shOverWrite);
Win.AdjustFrameCoords(27, 3, 53, 17);
...
```

Incorrect! All frame features are disposed when you adjust the frame coordinates. Always adjust the frame, then add your features.

See Also

AdjustWindow DeltaAdjustFrame

AdjustWindow

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure AdjustWindow(X1, Y1, X2, Y2 : Word); virtual;
```

Purpose

Set new window coordinates and adjust all related structures.

Description

Change the size or position (or both) of a window with this method. It works with either an inactive window or the current one. The coordinates specified are those of the active window rectangle. AdjustWindow automatically adjusts the positions of borders, headers, shadows, scroll bars and hot spots according to the applicable rules for each structure.

When AdjustWindow acts on the current window, it also updates the screen during the call. AdjustWindow allocates a temporary virtual screen and performs the update on that screen to avoid a flashing effect. Note that the virtual screen requires a temporary heap block of 2*ScreenWidth*ScreenHeight bytes. In order to retain the window's contents while it is resized, you must have either set the wSaveContents flag or else have an UpdateContents method in place to refill the window after it is resized. **Object Professional**'s higher level windows (PickList, EntryScreen, etc.) provide the appropriate UpdateContents methods.

If the window is on-screen but not current, AdjustWindow generates the epNonFatal + ecWinNotCurrent error and exits immediately. If the specified coordinates are invalid, it generates epNonFatal + ecWinCoordsBad. If there is insufficient heap space to reallocate the various dynamic structures, it generates epFatal + ecOutOfMemory.

ResizeWindow, MoveWindow, Zoom, and Unzoom generate calls to AdjustWindow.

Example

```
var
  Xlo, Ylo, Xhi, Yhi : Byte;
begin
  Win.Init(30, 5, 50, 15);
  Win.wOptionsOn(wBordered);
  Win.wFrame.AddScrollBar(frRR, 0, 100, DefaultColorSet);
  Win.Draw;
  Win.wFastText('Sample Text', 1, 1);
  {...Use window...}
  Win.Coordinates(Xlo, Ylo, Xhi, Yhi);
  Win.AdjustWindow(Xlo-10, Ylo+5, Xhi+10, Yhi+5);
  Result := Win.GetLastError;
  if Result <> 0 then
    WriteLn('Failed to adjust window, Result = ', Result)
  else
    {...Continue with window...};
end.
```

Win.Coordinates returns the active window coordinates and AdjustWindow sets new coordinates to make the window 20 characters wider and 10 taller.

See Also

MoveWindow
ResizeWindow
Unzoom
Zoom

Declaration

```
function AltFramePtr : AbstractFramePtr; virtual;
```

Purpose

Return a pointer to frame used when window is active but not current.

Description

AltFramePtr returns the address of the aFrame field of the RawWindow object. This frame is used only when the wAltFrame flag is set.

Using the AltFramePtr method rather than directly referring to the aFrame field is recommended to get the best future benefits of OOP. At this time, however, referring to aFrame gives the same results and is more efficient.

Because AltFramePtr returns a pointer of type AbstractFramePtr, it must be typecast as shown in the example following.

Example

```
FramePtr(Win.AltFramePtr)^.AddHeader('Standard Header', heTC);
```

Adds a header to Win's alternate frame. The header will be in the top center of the frame (heTC). In the absence of polymorphism, this statement is equivalent to

```
Win.aFrame.AddHeader('Standard Header', heTC);
```

See Also

MainFramePtr

ChangeAllScrollBars

Declaration

```
{$IFDEF UseScrollBars}
procedure ChangeAllScrollBars(MinHoriz, MaxHoriz : LongInt;
                              MinVert, MaxVert : LongInt); virtual;
```

Purpose

Change user ranges of existing scroll bars.

Description

This method changes all scroll bars of both the main frame and the alternate frame, if any. (Note the assumption that the main and alternate frames have identically-dimensioned scroll bars if they have any at all.) Bars on both the left and right edges of the frame get the new MinVert and MaxVert limits, and bars on the top and bottom edges get MinHoriz and MaxHoriz. Do not use this method if independent scroll bars are situated on opposing edges of the frame. Instead use the ChangeScrollBar method, which updates a single specified scroll bar.

If the window is current, ChangeAllScrollBars erases each associated slider. Otherwise it sets an internal flag which indicates that the user value associated with the slider is no longer valid. The sliders can be reinitialized and redrawn with DrawAllSliders.

If the window has no scroll bars, ChangeAllScrollBars does nothing.

Example

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.wFrame.AddScrollBar(frRR, 0, 100, DefaultColorSet);
Win.wFrame.AddScrollBar(frBB, 0, 100, DefaultColorSet);
Win.Draw;
{...Use the window...}
Win.ChangeAllScrollBars(-100, 100, -100, 100);
Win.DrawAllSliders(0, 0);
{...Continue with window...}
```

Both the horizontal and the vertical scroll bars are initially defined with ranges of 0 to 100. We later change these to -100 to 100. The old sliders are automatically erased, then DrawAllSliders draws the new ones.

See Also

ActiveFramePtr
ChangeScrollBar
DrawAllSliders

Declaration

```
procedure ChangeHeader(Index : Byte; S : string);
```

Purpose

Change header string, and if window is current, update screen.

Description

ChangeHeader modifies headers only on the main frame, wFrame. To modify a header on the alternate frame, aFrame, use the Frame.ChangeHeaderString method.

Index is the sequence number of the header to change. The first header added has sequence number 0, the second header 1, and so on. Typically, the index of a specific header will be stored when the header is added to the frame, using Frame.GetLastHeaderIndex. S is the new header string.

If the window is not current at the time of the call, the changed header will not appear until the window is made current (by calling Draw or StackWindow.Select).

ChangeHeader can return the same error codes as Frame.ChangeHeaderString.

Examples

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.wFrame.AddHeader('Standard Header', heTC);
Win.wFrame.AddHeader('Bottom Header', heBC);
Win.Draw;
{...Use window...}
Win.ChangeHeader(1, 'New Bottom Header');
{...continue with window...}
```

While the window is current the 'Bottom Header' is changed to 'New Bottom Header' and the screen is updated to show this change.

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.aFrame.AddHeader('Standard Header', heTC);
Win.Draw;
{...Use window...}
Win.ChangeHeader(0, 'New Header');
...
```

Incorrect! So far, this window has only defined aFrame headers. ChangeHeader only works on wFrame headers. Win.GetLastError will return epNonFatal+ecWinBadIndex.

See Also

Frame.ChangeHeaderString

ChangeScrollBar / Clear

Declaration

```
{$IFDEF UseScrollBars}
procedure ChangeScrollBar(Posn : FrameEdgeType; MinUser, MaxUser : LongInt);
```

Purpose

Change user range of specified scroll bar.

Description

This method changes the scroll bar on frame edge Posn of both the main and alternate frames. (Note the assumption that the main and alternate frames have identically-dimensioned scroll bars if they have any at all.) The new minimum and maximum limits are assigned to the scroll bar and, if the window is current, the associated slider is erased. Otherwise it sets an internal flag which indicates that the user value associated with the slider is no longer valid. The slider can be reinitialized and redrawn with DrawSlider.

If the window has no scroll bar on the specified edge, ChangeScrollBar does nothing.

Example

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.wFrame.AddScrollBar(frRR, 0, 100, DefaultColorSet);
Win.Draw;
{...Use the window...}
Win.ChangeScrollBar(frRR, -100, 100);
Win.DrawSlider(frRR, 0);
{...Continue with window...}
```

Changes the range of the scroll bar from its initial 0 to 100 to -100 to 100. The current slider is automatically erased, then DrawSlider draws the new one.

See Also

ChangeAllScrollBars DrawSlider

Declaration

```
procedure Clear; virtual;
```

Purpose

Clear active window area with wTextXxx and wBackChar.

Description

Uses the character wBackChar and either wTextColor or wTextMono (depending on video mode) to fill the active window rectangle. Also moves the cursor to the upper left corner of the window.

The window must be current at the time Clear is called or it will generate an error (epNonFatal + ecWinNotCurrent).

By default, wBackChar is a space and this routine acts much like OPCRT.ClrScr. Use SetBackChar to choose an alternate background character.

Example

```
Win.SetBackChar(#178);
Win.Clear;
```

Clears Win's active window with the "heavy" background fill character and homes the cursor.

See Also

ClearContents SetBackChar

Declaration
```
procedure ClearContents;
```

Purpose
Clear region within frame with wTextXxx and wBackChar.

Description
Works like Clear, except that the rectangle cleared is the largest region that will fit inside of the window's border. This rectangle differs from the active window rectangle only if AdjustFrameCoords has been used.

As with Clear, ClearContents moves the cursor to the upper left corner of the active window rectangle.

Example
```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.AdjustFrameCoords(25, 3, 55, 17);
Win.Draw;
Win.ClearContents;
```

Clears the rectangle (26, 4, 54, 16) and moves the cursor to the absolute (X, Y) coordinate (30, 5). Win.Clear would clear the rectangle (30, 5, 50, 15) and move the cursor to the same place.

See Also
Clear

Declaration
```
procedure ClearErrors;
```

Purpose
Clear all pending errors.

Description
Calls GetLastError (a virtual method) until it returns zero. This is especially useful in the case of objects derived from AbstractWindow since some of them collect errors from several sources: the associated Frames and ScreenRects, the RawWindow, and the CommandWindow, at least.

ClearErrors is actually implemented in the AbstractWindow object.

Example
```
Win.ClearErrors;
```

Clears all pending result codes for Win and its window components.

See Also
GetLastError

Coordinates / CursorInWindow

Declaration
```
procedure Coordinates(var X1, Y1, X2, Y2 : Byte);
```

Purpose
Return the active coordinates of the window.

Description
The returned values are the absolute screen coordinates of the active window rectangle. These are the same values passed to the window's constructor; they will also track calls to AdjustWindow, ResizeWindow, MoveWindow, Zoom, and Unzoom.

Example
```
var Xlo, Ylo, Xhi, Yhi : Byte;
...
Win.Coordinates(Xlo, Ylo, Xhi, Yhi);
```
Returns the absolute screen coordinates of Win's active screen rectangle.

See Also
Extent Frame.WithinFrameCoords

Declaration
```
function CursorInWindow : Boolean;
```

Purpose
Return True if cursor is currently within active window.

Description
Calls OPCRT.WhereXYAbs to read the current cursor position, then calls InWindow to determine whether that position is within the active window rectangle.

Example
```
Win.Init(30, 5, 50, 15);
Win.Draw;
{...Use window...}
if Win.CursorInWindow then
  WriteLn('in window')
else
  Writeln('not in window');
```
Reports whether or not the cursor is in the current window.

See Also
InWindow WhereXYAbs (OPCRT)

Declaration

```
procedure DeltaAdjustFrame(dX1, dY1, dX2, dY2 : Integer);
```

Purpose

Change frame coordinates of an inactive window. Reallocates wCovers buffer but changes nothing else.

Description

The existing frame coordinates are adjusted by the specified relative amounts: the frame's lower X limit by dX1, the upper X limit by dX2, and so on. The internal frame coordinates (both wFrame and aFrame) are set to the new values and the ScreenRect that stores the covered region is reallocated. No other frame features--header positions, scroll bar extents, shadows--are adjusted.

The window must not be visible. If it is, DeltaAdjustFrame generates an error (epNonFatal + ecWinIsActive) and exits immediately. Other possible errors include (epNonFatal + ecWinCoordsBad) and (epFatal + ecOutOfMemory).

This method provides a low level hook that is useful for developing a "paned window" manager (like that used by Turbo Debugger). This kind of window manager tiles unbordered panes into an enclosing bordered rectangle which also manages an irregular frame separating the panes. Unless you are doing this, you're unlikely to need DeltaAdjustFrame.

See Also

AdjustFrameCoords AdjustWindow

Declaration

```
destructor Done; virtual;
```

Purpose

Destroy a window.

Description

If the window is active at the time Done is called, it is first erased. Then all associated buffers and frame components are deallocated.

Example

```
Win.Done;
```

Disposes of all Win components.

See Also

Erase

Draw

Declaration

```
procedure Draw; virtual;
```

Purpose

Draw or update a window.

Description

The behavior of Draw depends on whether the window is already active at the time it is called. If the window is inactive, Draw performs the following actions in sequence:

- saves current cursor position and shape
- saves OPCRT window coordinates and default text attribute
- saves mouse window coordinates, cursor position, and mouse visibility if wSetMouse or wFullMouseWindow is set
- deactivates the current window, if any. This involves saving its cursor position, updating its contents buffer if wSaveContents is set, and displaying its alternate frame if the wAltFrame flag is set
- sets the new OPCRT window coordinates and default text attribute
- points CurrentWindow to the window (by calling the MarkCurrent virtual method)
- positions the cursor
- saves the underlying screen
- marks the window active
- calls the open procedure
- if available, restores previous contents or, if wClear is set, clears the window
- calls wFrame's Draw method
- if wUserContents is set, calls the UpdateContents virtual method
- sets mouse window if wFullMouseWindow is enabled or sets mouse window, mouse cursor position, and mouse visibility if wSetMouse is set

If the window is already active but not current, it performs only the following steps:

- deactivates the current window, if any. This involves saving its cursor position, updating its contents buffer if wSaveContents is set, and displaying its alternate frame if the wAltFrame flag is set
- sets the new OPCRT window coordinates and default text attribute
- points CurrentWindow to the window (by calling the MarkCurrent virtual method)
- positions the cursor
- erases shadows, if any, from the alternate frame if the wAltFrame flag is set
- if wUserContents is set, calls the UpdateContents virtual method
- sets mouse window if wFullMouseWindow is enabled or sets mouse window, mouse cursor position, and mouse visibility if wSetMouse is set

To summarize, Draw makes the window visible on the screen, positions the cursor, and sets the OPCRT window coordinates so that Write statements will appear within the window.

Example

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.Draw;
WriteLn('In upper left corner of window');
Win.Done;
```

Draws a small, bordered window, writes some text to the upper left corner, and erases the window.

See Also

Erase StackWindow.Draw

Declaration

```
{$IFDEF UseScrollBars}
procedure DrawAllSliders(HorizVal, VertVal : LongInt); virtual;
```

Purpose

Draw or update all sliders.

Description

This method redraws all the sliders of the main frame if the window is current, or updates their values if not. The alternate frame's slider values are updated if the wAltFrame flag is set. (Note the assumption that the main and alternate frames have identically-dimensioned scroll bars if they have any at all.) When changes are not drawn immediately, they will show up the next time the frame is redrawn.

HorizVal specifies the user value for horizontal scroll bars on both top and bottom edges of the frame. VertVal specifies the value for left and right. If the specified values match the last ones for which sliders were drawn, no action is taken.

Do not use this method if independent scroll bars are situated on opposing edges of the frame. Instead use the DrawSlider method, which updates a single specified scroll bar.

If the window has no scroll bars, DrawAllSliders does nothing.

Example

```
Win.ChangeAllScrollBars(-100, 100, -100, 100);
Win.DrawAllSliders(0, 0);
```

Changes the range on both horizontal and vertical scroll bars to -100 to 100, then draws all sliders. The sliders will be centered on both scroll bars.

See Also

ChangeAllScrollBars DrawSlider

DrawSlider

Declaration

```
{$IFDEF UseScrollBars}
procedure DrawSlider(Posn : FrameEdgeType; UserVal : LongInt);
```

Purpose

Draw or update the slider.

Description

This method redraws the slider on the specified edge of the main frame if the window is current, or updates its value if not. The alternate frame's slider value is updated if the wAltFrame flag is set. (Note the assumption that the main and alternate frames have identically-dimensioned scroll bars if they have any at all.) When changes are not drawn immediately, they will show up the next time the frame is redrawn.

If the slider is already visible and the specified UserVal matches the previous value, no action is taken.

If the window has no scroll bar on the specified edge, DrawSlider does nothing.

Example

```
SliderPos := 1;
Finished := False;
repeat
  {...Wait for a mouse press...}
  Win.EvaluateMousePos;
  UserValue := Win.PosResults(FramePos, HotCode);
  case FramePos of
    frOutsideFrame : {Mouse outside of frame};
    frInsideActive : {Mouse inside of window};
  else
    case HotCode of
      hsDecV : begin  {Mouse on decrement arrow of scroll bar}
                 if SliderPos > 1 then
                   Dec(SliderPos);
                 Win.DrawSlider(frRR, SliderPos);
               end;
      hsIncV : begin  {Mouse on increment arrow of scroll bar}
                 if SliderPos < 100 then
                   Inc(SliderPos);
                 Win.DrawSlider(frRR, SliderPos);
               end;
    end;
  end;
until Finished;
```

Waits for mouse presses and checks the current location of the mouse. If the mouse is over a scroll bar hot spot, the SliderPos variable is adjusted and the slider is re-drawn.

Declaration

```
procedure EnableAlternateOpen(OpenProc, CloseProc : RawWindowProc;
                              StageDelay : Word);
```

Purpose

Enable alternate draw and erase routines.

Description

EnableAlternateOpen assigns user-defined procedures to be called when the window is opened and closed. OPWINDOW defines two such procedures which create exploding window effects–ExplodeOpenProc and ExplodeCloseProc–and a third that does nothing–EmptyOpenCloseProc. You can define additional procedures to create other special effects.

Procedures passed as parameters to this routine must be global in scope, declared using the FAR model, and must match the RawWindowProc type. These procedures may not call the RawWindow.Draw or RawWindow.Erase routines; this would lead to uncontrolled recursion and a program crash. Instead they should call more primitive routines from the Frame object or the OPCRT unit. The routines are passed a pointer to the RawWindow being drawn or erased which they can use to get needed information about the window. See ExplodeOpenProc in OPWINDOW.IN1 for an example.

The StageDelay parameter is interpreted by ExplodeOpenProc and ExplodeCloseProc as the number of milliseconds to delay between stages in the explosion. The value passed to EnableAlternateOpen is stored in the wStageDelay field of the window.

The most common use of EnableAlternateOpen may be to create exploding windows that close in a normal manner rather than imploding. See the second example following.

If alternate open/close routines are being used and a window object is to be stored in a stream, it is necessary to register the user-defined procedures with the stream manager. Use the pointer codes ptExplodeOpenProc and ptExplodeCloseProc for this purpose. See ExplodingWindowStream in OPWINDOW.IN1 for a specific example.

Examples

```
uses
  OpCrt, OpFrame, OpWindow;
var
  Win : RawWindow;

  {$F+}
  procedure MyOpenProc(WPtr : RawWindowPtr);
    {-flash some background patterns}
  const
    FillChar : array[1..4] of char = (#178, #177, #176, ' ');
  var
    I : Byte;
  begin
    with WPtr^ do begin
      wFrame.DrawBorder;
      for I := 1 to 4 do begin
        ClearWindow(wXL, wYL, wXH, wYH, FillChar[I], TextAttr);
        Delay(wStageDelay);
      end;
    end;
  end;

  procedure MyCloseProc(WPtr : RawWindowPtr);
    {-flash some background patterns}
  const
    FillChar : array[1..4] of char = (#178, #177, #176, ' ');
  var
    I : Byte;
```

```
    begin
      with WPtr^ do begin
        for I := 4 downto 1 do begin
          ClearWindow(WXL, WYL, WXH, WYH, FillChar[I], TextAttr);
          Delay(wStageDelay);
        end;
      end;
    end;
    {$F-}

  begin
    Win.InitCustom(30, 5, 50, 15, DefaultColorSet,
                   DefWindowOptions or wBordered);
    Win.EnableAlternateOpen(MyOpenProc, MyCloseProc, 100);
    Win.Draw;
    Delay(1000);
    Win.Erase;
  end.
```

Uses custom window open/close procedures to create a special effect. These procedures simply display a changing background as the windows open and close.

```
Win.EnableAlternateOpen(ExplodeOpenProc, EmptyOpenCloseProc, 25);
```

Sets up a window that explodes when opened but closes normally.

See Also

Draw EnableExplosions

Declaration
```
procedure EnableExplosions(StageDelay : Word);
```

Purpose
Enable exploding windows with StageDelay milliseconds per stage.

Description
This routine assigns ExplodeOpenProc and ExplodeCloseProc to the corresponding procedure variables in the RawWindow object. StageDelay is also stored in a data field, to be referenced by the two procedures when explosions are in progress.

The algorithm used to produce the exploding effect is more easily experienced than explained. Suffice it to say that a series of frames is drawn, starting with a minimum size rectangle and expanding smoothly to the final frame size. If the contents buffer of the window is available and initialized, the window's contents are restored at each stage of the explosion. A delay of StageDelay milliseconds is generated after each stage. The final frame embellishments–headers, shadows, etc.–are not drawn until the explosion is complete. The reverse process occurs when the window is closed.

Explosions will be accompanied by sound effects if the window's wSoundEffects flag is set.

If explosions are enabled and a window object is to be stored in a stream, it is necessary to register the explosion-related procedures with the stream manager. To do so, pass the ExplodingWindowStream procedure as a parameter to the stream's RegisterHier method:

```
S.RegisterHier(ExplodingWindowStream);
```

Example
```
Win.InitCustom(30, 5, 50, 15, DefaultColorSet, DefWindowOptions or wBordered);
Win.EnableExplosions(20);
Win.Draw;
```

Create an exploding window with a 20 millisecond delay between stages.

See Also
EnableAlternateOpen IdStream.RegisterHier

Declaration
```
procedure EnableNormalOpen;
```

Purpose
Disable exploding windows or alternate draw routines.

Description
Sets the RawWindow's procedure pointers back to EmptyOpenCloseProc, thereby disabling any special effects.

Example
```
Win.EnableExplosions(20);
Win.Draw;
{...Use Window...}
Win.Erase;
Win.EnableNormalOpen;
Win.Draw;
{...Use Window again...}
```

This window is first set up to use exploding windows. Later in the program, the exploding window feature is turned off.

See Also
EnableAlternateOpen

Erase / EraseSlider

Declaration

```
procedure Erase; virtual;
```

Purpose

Erase a window.

Description

If the window is not active at the time of the call, Erase does nothing. Otherwise it performs the following actions:

- saves current cursor position and shape
- saves mouse window/position/visibility if wSetMouse is set
- saves the window's contents if wSaveContents is set
- erases the window's shadows if any
- calls the close procedure
- restores the underlying screen
- marks the window as inactive
- sets the CurrentWindow pointer to nil if CurrentWindow is being erased
- restores mouse window/position/visibility if wSetMouse or wFullMouseWindow is set
- restores cursor position and shape

To summarize, Erase restores the screen to its state prior to when the window was drawn.

Note that Erase does not dispose of the window. Heap space remains allocated for the various window data structures. The window can be drawn again simply by calling Draw.

Example

```
Win.Draw;
{...Use Window...}
Win.Erase;
```

Erase the window after using it. The window still exists and can be drawn again later.

See Also

Done Draw

Declaration

```
{$IFDEF UseScrollBars}
procedure EraseSlider(Posn : FrameEdgeType);
```

Purpose

Erase the slider.

Description

This method erases the slider on the specified edge of the main frame if the window is current, or marks it invisible if not. The alternate frame's slider flag is also updated if the wAltFrame flag is set. (Note the assumption that the main and alternate frames have identically-dimensioned scroll bars if they have any at all.) When changes are not drawn immediately, they will show up the next time the frame is redrawn.

If the window has no scroll bar on the specified edge, EraseSlider does nothing.

You typically don't need to call EraseSlider. The DrawSlider method automatically erases the previous slider before drawing a new one.

Example

```
Win.EraseSlider(frRR); Win.EraseSlider(frBB);
```

Erases the current window sliders without drawing new ones.

See Also

DrawAllSliders DrawSlider

Declaration
```
procedure Error(Code : Word); virtual;
```

Purpose
Report an error.

Description
Applications do not call this routine directly. When a RawWindow method detects an error, it calls the Error method to report it. The default implementation of this method simply assigns the error code to an internal status variable. An object derived from RawWindow may override the Error method to provide different error handling behavior, or may call Error to store an error code.

Example
```
Win.Error(MyErrorCode);
```
Stuff an application error code into Win's error field.

See Also
GetLastError

Declaration
```
{$IFDEF UseMouse}
procedure EvaluateMousePos;
```

Purpose
Evaluate current mouse position as with EvaluatePos.

Description
If a mouse is installed, EvaluateMousePos calls OPMOUSE.MouseWhereXYabs to get the mouse cursor position, then calls EvaluatePos to evaluate that position relative to the window's frame. See EvaluatePos for more information.

Note that EvaluateMousePos makes no check to determine whether the mouse cursor and window are visible.

Example
```
var
  UserVal : LongInt;
  FramePos : FramePosType;
  HotCode : Byte;
...
{...Init/Draw window...}
Finished := False;
repeat
  Win.EvaluateMousePos;
  UserVal := Win.PosResults(FramePos, HotCode);
  Fastwrite(CharStr(' ', ScreenWidth), 25, 1, $07);
  Fastwrite(FPName[FramePos], 25, 1, $07);
  Fastwrite(Long2Str(HotCode), 25, 20, $07);
  Fastwrite(Long2Str(UserVal), 25, 40, $07);
until Finished;
Win.Done;
```
As a mouse is moved around the screen, report its relation to Win's frame. See EvaluatePos for a similar (but complete) program.

See Also
EvaluatePos PosResults

EvaluatePos

Declaration

```
procedure EvaluatePos(X, Y : Byte);
```

Purpose

Evaluate specified absolute position relative to window, hotspots, and scrollbars.

Description

EvaluatePos performs a series of comparisons to determine where the specified position falls relative to the window. It stores up to three different indicators as fields within the object; these fields are accessed with the PosResults method. The three indicators are: the frame position (is the point outside of the frame, inside the frame, or on the frame?); the hot spot code, if the point is within a hot spot; and the scroll bar user value, if it is within a scroll bar.

If the window is current or has no alternate frame, EvaluatePos uses the main frame as its coordinate reference. Otherwise it uses the alternate frame. Note that EvaluatePos makes no check to determine whether the window is active; it will return the same results even if the window isn't visible.

Example

```
const FPName : array[FramePosType] of string[14] =
        ('frTL', 'frBL', 'frTR', 'frBR','frTT', 'frBB', 'frLL', 'frRR',
         'frInsideActive', 'frInsideFrame', 'frOutsideFrame');
var Win : RawWindow;
    X, Y, HotCode : Byte;
    FramePos : FramePosType;
    UserCode : LongInt;
    Finished : Boolean;
begin
  Win.InitCustom(30, 5, 50, 15, DefaultColorSet, DefWinOptions or wBordered);
  Win.wFrame.AddScrollBar(frRR, 0, 1000, DefaultColorSet);
  Win.wFrame.AddScrollBar(frBB, -100, 100, DefaultColorSet);
  Win.wFrame.AddHotSpot(frTL, #254, DefaultColorSet);
  Win.wFrame.AddHotBar(frTT, hsRegion0);
  Win.Draw;
  X := 1; Y := 1; Finished := False;
  repeat
    GotoXYAbs(X, Y);
    Win.EvaluatePos(Y, Y);
    UserCode := Win.PosResults(FramePos, HotCode);
    Fastwrite(CharStr(' ', ScreenWidth), 25, 1, $07);
    Fastwrite(FPName[FramePos], 25, 1, $07);
    Fastwrite(Long2Str(HotCode), 25, 20, $07);
    Fastwrite(Long2Str(UserCode), 25, 40, $07);
    case ReadKeyWord of
      $4800 : if Y > 1 then Dec(Y);
      $4B00 : if X > 1 then Dec(X);
      $4D00 : if X < ScreenWidth then Inc(X);
      $5000 : if Y < ScreenHeight then Inc(Y);
      $011B : Finished := True;
    end;
  until Finished;
  Win.Done;
end.
```

Demonstrates how EvaluatePos and PosResults work together. The program constantly displays the relationship between the cursor position and Win's frame.

See Also

EvaluateMousePos
PosResults
WindowStack.EvaluateStackPos

Declaration

```
procedure Extent(var X1, Y1, X2, Y2 : Byte; IncludeShadows : Boolean);
```

Purpose

Return the complete extent of the window, counting shadows if requested.

Description

The "extent" means the coordinates of a rectangle that completely encloses all components of the window. Structures checked include the active window region, the frame (both main and alternate), the border and headers, and optionally the shadows. This information is useful in determining whether a particular window coordinate adjustment is valid.

Shadows typically don't set a real limit while drawing or sizing a window because OPWINDOW will clip them automatically.

Example

```
var Xlo, Ylo, Xhi, Yhi : Byte;
...
Win.Extent(Xlo, Ylo, Xhi, Yhi, True);
```

Returns the absolute screen coordinates of the tallest and widest extent of Win, including shadows.

See Also

Coordinates

Frame-relative Writing

Declarations

```
procedure fFastWrite(St : string; Row, Col, Attr : Byte);
  {-Write a string}
procedure fFastText(St : string; Row, Col : Byte);
  {-Write St at Row,Col without changing the underlying video attribute}
procedure fFastVert(St : string; Row, Col, Attr : Byte);
  {-Write a string vertically}
procedure fFastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
  {-Fill Number chars at Row,Col in Attr}
procedure fChangeAttribute(Number : Word; Row, Col, Attr : Byte);
  {-Change Number video attributes to Attr starting at Row,Col}
procedure fFastCenter(St : string; Row, Attr : Byte);
  {-Write St centered on Row in Attr}
procedure fFastFlush(St : string; Row, Attr : Byte);
  {-Write St flush right on Row in Attr}
procedure fFastRead(Number, Row, Col : Byte; var St : string);
  {-Read Number characters from the screen into St starting at Row,Col}
procedure fReadAttribute(Number, Row, Col : Byte; var St : string);
  {-Read Number attributes from the screen into St starting at Row,Col}
procedure fWriteAttribute(St : String; Row, Col : Byte);
  {-Write string of attributes St at Row,Col without changing characters}
procedure fFlexWrite(St : string; Row, Col : Byte; var FAttrs : FlexAttrs);
  {-Write a string flexibly using window-relative coordinates}
procedure fFastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
  {-Write St at Row,Col in Attr (video attribute) without snow.
    Control characters displayed in Ctrl as upper-case letters}
```

Purpose

Write using coordinates relative to the frame boundary.

Description

These routines differ from the window-relative write routines (wFastWrite, etc.) only if you've called AdjustFrameCoords to set non-default frame coordinates. If so, they provide convenient methods to access the screen relative to the frame boundaries rather than the window boundaries.

If the frame has a border, the (1, 1) position for these routines is just inside the upper left hand corner of the border. If the frame has no border, the (1, 1) position is coincident with the upper left corner of the frame.

These routines clip at the frame boundaries when appropriate. They will not allow you to write or read outside of the frame.

See Window-relative Writing for more details about each method.

Declaration

```
function GetCursor : CursorType;
```

Purpose

Return current cursor shape.

Description

GetCursor accesses an internal data field of the object to return the window's current cursor shape. This data field is initialized to DefCursor (typically the normal underline cursor, cuNormal) when the window is initialized. It is updated when SetCursor is called, when the window is erased, and when another window is drawn. Note that calling OPCRT.SetCursorSize does not directly update the data field.

CursorType is an enumerated type describing the typical shapes a cursor can have (cuNormal, cuBlock, etc.). If the cursor has been set to a non-standard shape, GetCursor will return cuUnknown.

Example

```
var
  SaveCursor : CursorType;
...
  Win.Init(30, 5, 50, 15);
  SaveCursor := Win.GetCursor;
  Win.SetCursor(cuBlock);
  Win.Draw;
  {...Use Window...}
  Win.SetCursor(SaveCursor);
```

Saves the current cursor shape, changes it to a block cursor, then later restores it to its original shape.

See Also

CursorType (OPCRT) SetCursor

GetLastError / Height

Declaration

```
function GetLastError : Word; virtual;
```

Purpose

Return and clear the last error code.

Description

Returns zero if no error is pending, otherwise one of the error codes listed in Appendix D.

GetLastError accesses the error fields of several objects: the RawWindow itself as well as those of the objects that it contains. While it's not likely, there may be more than one error pending at once. GetLastError clears only one error at a time, so a subsequent call may return non-zero. To guarantee that GetLastError will return zero, call the ClearErrors method.

It's not necessary to call GetLastError after every call to a RawWindow routine, but it doesn't hurt to do so. Documentation for each method indicates when errors are possible.

GetLastError doesn't return window errors in one special case: after a constructor fails. In that case, the GetLastError method (and the data fields it references) aren't even guaranteed to exist. Use the global variable InitStatus (from the OPROOT unit) instead.

Examples

```
Win.wFrame.AddScrollBar(frRR, 0, 1000, DefaultColorSet);
Result := Win.GetLastError;
if Result <> 0 then
  WriteLn('Failed to add window feature, Result = ', Result);
```

Check for an error after adding a scroll bar. It is especially important to check after calling the various Add methods from the Frame object, since these must typically allocate heap space and verify coordinate values.

```
if Win.GetLastError <> 0 then
  WriteLn('Failed to add window feature, Result = ', Win.GetLastError);
```

Incorrect! Referencing GetLastError clears it to zero.

See Also

ClearErrors Error

Declaration

```
function Height : Word;
```

Purpose

Return the active height of the window.

Description

Returns the number of rows in the active window rectangle.

This method is actually implemented in the AbstractWindow object.

Examples

```
Win.wFrame.AddCustomShadow(#178, frBB, 1, 1, Win.Width+2, 1, $02, $70);
Win.wFrame.AddCustomShadow(#178, frRR, 1, 1, 1, Win.Height+2, $02, $70);
```

Uses the Height function to add a shadow the same height as the frame (2 rows larger than the window).

```
Win.wFastText('bottom of window', Win.Height, 1);
```

Writes a line of text to the bottom row of the window.

See Also

Coordinates Width

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte);
```

Purpose

Create a window with default options and colors.

Description

Init instantiates a RawWindow using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if not Win.Init(30, 5, 50, 15) then begin
  Writeln('Failed to Init RawWindow, Status = ', InitStatus); Halt;
end;
```

The upper left corner of the window will be at screen location (30,5); the lower right corner will be at screen location (50,15). The default options (unless changed) do not include the flag that produces a window border.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt);
```

Purpose

Create a window with custom options.

Description

This constructor instantiates a RawWindow using the specified window Options and Colors. X1, Y1, X2, and Y2 are the absolute coordinates for the interior portion of the window, where text will be written and scrolled.

InitCustom allocates a buffer to store the screen region the window will later cover. If the wBordered option is not set, this buffer uses 2*(X2-X1+1)*(Y2-Y1+1) bytes. If wBordered is set, it uses 2*(X2-X1+3)*(Y2-Y1+3) bytes. If the wSaveContents flag is set, another buffer of approximately the same size is allocated to store the contents of the window.

InitCustom reads the TextColor and TextMono fields of the ColorSet, as well as the fields used by the Frame object.

If InitCustom fails, the error code will be stored in InitStatus. Possible failure modes include bad window coordinates (epFatal+ecWinCoordsBad) and insufficient memory (epFatal+ecOutOfMemory).

See the description of the window options at the start of this section for more information.

Example

```
var
  Win : RawWindow;
  WinOpts : LongInt;
begin
  WinOpts := wBordered+wClear+wSaveContents+wResizeable;
  if not Win.InitCustom(30, 5, 50, 15, DefaultColorSet, WinOpts) then begin
    Writeln('Failed to Init RawWindow, Status = ', InitStatus);
    Halt;
  end;
```

Instantiates a framed, resizable window. The screen will be cleared when it is first displayed. The contents of the window will be saved when the window is erased. The window will use default colors.

See Also

Frame.InitCustom Init

InWindow / IsActive / IsCurrent

Declaration
```
function InWindow(AX, AY : Byte) : Boolean;
```

Purpose
Return True if specified absolute position is within active window.

Description
InWindow compares AX and AY to the active window coordinates and returns True if the position is within the window. Note that InWindow does not check to see whether the window is visible at the time.

Example
```
if Win.InWindow(MouseWhereXabs, MouseWhereYabs) then
  WriteLn('mouse cursor in window')
else
  Writeln('mouse cursor not in window');
```
Reports whether the mouse cursor is currently within the window Win.

See Also
CursorInWindow EvaluatePos

Declaration
```
function IsActive : Boolean; virtual;
```

Purpose
Return True if window is on-screen.

Description
Checks an internal flag which is set when the window is drawn and reset when the window is erased.

Example
```
if not Win.IsActive then
  Win.Draw;
```
If Win is not currently on the screen then draw it.

Declaration
```
function IsCurrent : Boolean; virtual;
```

Purpose
Return True if window is the current one.

Description
The current window is the one most recently drawn. The global variable CurrentWindow always contains a pointer to the current window. IsCurrent compares the address of the window object to CurrentWindow and if they are the same returns True.

Example
```
if not Win.IsCurrent then
  Win.Draw;
WriteLn('this output appears within Win');
```
If Win is not current, Draw makes it so. For a RawWindow, Draw is the appropriate method to call to make a window current, even if the window is already on-screen but not current. After making the window current, the output from Write and WriteLn statements will appear within the window.

Declaration

```
{$IFDEF UseAdjustableWindows}
function IsZoomed : Boolean; virtual;
```

Purpose

Return True if window is currently zoomed.

Description

Checks an internal flag which is set when Zoom is called and reset when Unzoom executes. If the window is zoomed, its size is guaranteed to be the maximum available to the window.

We often use IsZoomed to determine whether to call Zoom or Unzoom when a toggle key is pressed.

Examples

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
Win.Draw;
Win.Zoom;
if Win.IsZoomed then
  Win.wFastText('This line is too long for an unzoomed window', 1, 1)
else
  Win.wFastText('This line just fits', 1, 1);
```

Checks the zoom state of Win before deciding how long a line it can write.

```
if Win.IsZoomed then
  Win.Unzoom
else
  Win.Zoom;
```

Toggles the zoom state of a window.

See Also

Zoom

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a RawWindow from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to RawWindow.Store. Load reads the fixed data fields of the RawWindow and calls the Load methods of the component objects of the window, including Frame and ScreenRect. If the window's wStoreContents flag has been set and the window's contents rectangle was initialized at the time the stream was written, Load also reads the contents of the window.

The stream registration procedure for a RawWindow is RawWindowStream. If the window uses explosion special effects, the stream registration procedure ExplodingWindowStream must also be called.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  Win : RawWindow;
  S : BufIdStream;
begin
  S.Init('RAWWIN.STM', SOpen, 1024);
  S.RegisterHier(RawWindowStream);
  S.Get(Win);
  if S.GetStatus <> 0 then
    Halt;
  S.Done;

  Win.Draw;
  Delay(1000);
  Win.Done;
end.
```

Load Win from a stream created by the Store example using the stream's Get method, which calls RawWindow.Load indirectly.

You'll probably want to put in more error checking code than this example shows. See the BufIdStream example for proper error checking when reading/writing streams.

See Also

Frame.Load
ScreenRect.Load
Store

Declaration
```
function MainFramePtr : AbstractFramePtr; virtual;
```

Purpose
Return a pointer to frame used when window is current.

Description
MainFramePtr returns the address of the wFrame field of the RawWindow object.

Using the MainFramePtr method rather than directly referring to the wFrame field is recommended to get the best future benefits of OOP. At this time, however, referring to wFrame gives the same results and is more efficient.

Because MainFramePtr returns a pointer of type AbstractFramePtr, it must be typecast as shown in the example following.

Example
```
FramePtr(Win.MainFramePtr)^.AddHeader('Standard Header', heTC);
```

Adds a header to Win's main frame. The header will be in the top center of the frame (heTC). In the absence of polymorphism, this statement is equivalent to

```
Win.wFrame.AddHeader('Standard Header', heTC);
```

See Also
ActiveFramePtr AltFramePtr

Declaration
```
procedure MarkCurrent; virtual;
```

Purpose
Called when window is made current.

Description
Applications do not call this routine directly. RawWindow.Draw calls this method, which by default just assigns the address of the window to the global variable CurrentWindow. An object derived from RawWindow may override the MarkCurrent method to perform additional actions, or may call MarkCurrent at appropriate times to update the CurrentWindow variable.

Declaration
```
procedure MarkNotCurrent; virtual;
```

Purpose
Called when window is deselected.

Description
Applications do not call this routine directly. Draw and Erase call this method, which by default does nothing. (Draw calls it only if there is already a current window which is being superseded by the newly drawn window.) An object derived from RawWindow may override the MarkNotCurrent method to perform additional actions.

MoveWindow

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure MoveWindow(DX, DY : Integer); virtual;
```

Purpose

Move a window.

Description

MoveWindow first assures that the window is not zoomed; if so, it generates a warning (epWarning + ecWinIsZoomed) and exits. If DX and DY are both zero, MoveWindow exits immediately to avoid wasting time. Otherwise, MoveWindow calls AdjustWindow effectively as follows:

```
var X1, Y1, X2, Y2 : Byte;
...
Coordinates(X1, Y1, X2, Y2);
AdjustWindow(X1+DX, Y1+DY, X2+DX, Y2+DY);
```

Note that the coordinate adjustments are relative to the window's active rectangle, not the frame. See AdjustWindow for further information.

Example

```
uses
  OpCrt, OpFrame, OpWindow;
var
  Win : RawWindow;
  Finished : Boolean;
begin
  Win.Init(30, 5, 50, 15);
  Win.wOptionsOn(wBordered);
  Win.Draw;
  Finished := False;
  repeat
    case ReadKeyWord of
      $4800 : Win.MoveWindow(0, -1);          {Up arrow}
      $4B00 : Win.MoveWindow(-1, 0);          {Left Arrow}
      $4D00 : Win.MoveWindow(1, 0);           {Right Arrow}
      $5000 : Win.MoveWindow(0, 1);           {Down arrow}
      $011B : Finished := True;               {Escape}
    end;
  until Finished;
  Win.Done;
end.
```

Demonstrates how to move a window in response to cursor commands.

See Also

AdjustWindow
ResizeWindow
Zoom

Declaration

```
function PosResults(var FramePos : FramePosType; var HotCode : Byte) : LongInt;
```

Purpose

Return values from the most recent call to EvaluatePos or EvaluateMousePos.

Description

PosResults simply accesses three data fields within the RawWindow object. These data fields must be initialized by calling EvaluatePos, EvaluateMousePos, WindowStack.EvaluateStackPos, or WindowStack.EvaluateStackMousePos first. These routines compare a specified (X, Y) position to the window's coordinates.

FramePos returns a number indicating where the position fell relative to the frame. It returns any of the following values:

```
frOutsideFrame          - outside of the frame
frInsideFrame           - inside of the frame but outside the active window
frInsideActive          - inside of the active window
frTL, frTR, frBL, frBR - on a corner of the frame
frTT, frBB, frLL, frRR - on an edge of the frame
```

HotCode returns the associated hot spot code if the position is within a hot spot. It returns hsNone if the position is not within a hot spot. See the constants in section B.2 (OPFRAME) for additional information.

The function result returns a scroll bar user value only if the position is within a scroll bar. Otherwise it returns zero. For example, if the position were one-quarter of the way down a vertical scroll bar, and the bar's limits were from -100 to 100, PosResults would return -50. Note that HotCode always returns the value hsBar when the PosResults function returns a meaningful value.

Example

See EvaluatePos.

See Also

EvaluateMousePos
EvaluatePos
WindowStack.EvaluateStackPos

ResizeWindow

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure ResizeWindow(DX, DY : Integer); virtual;
```

Purpose

Resize a window.

Description

ResizeWindow first assures that the window is not zoomed; if so, it generates a warning (epWarning+ecWinIsZoomed) and exits. Next it checks whether the wResizeable flag is set; if not, it generates an error (epNonFatal+ecWinNotSizeable) and exits. If DX and DY are both zero, ResizeWindow exits immediately to avoid wasting time. Otherwise, ResizeWindow calls AdjustWindow effectively as follows:

```
var
  X1, Y1, X2, Y2 : Byte;
...
  Coordinates(X1, Y1, X2, Y2);
  AdjustWindow(X1, Y1, X2+DX, Y2+DY);
```

Note that the coordinate adjustments are relative to the window's active rectangle, not the frame. See AdjustWindow for further information.

Example

```
uses
  OpCrt, OpFrame, OpWindow;
var
  Win : RawWindow;
  Finished : Boolean;
begin
  Win.Init(30, 5, 50, 15);
  Win.wOptionsOn(wBordered);
  Win.Draw;
  Finished := False;
  repeat
    case ReadKeyWord of
      $4800 : Win.ResizeWindow(0, -1);          {Up arrow}
      $4B00 : Win.ResizeWindow(-1, 0);          {Left Arrow}
      $4D00 : Win.ResizeWindow(1, 0);           {Right Arrow}
      $5000 : Win.ResizeWindow(0, 1);           {Down arrow}
      $011B : Finished := True;                 {Escape}
    end;
  until Finished;
  Win.Done;
end.
```

Demonstrates how to resize a window in response to cursor commands.

See Also

AdjustWindow
MoveWindow
Zoom

Declaration

```
procedure SaveContents; virtual;
```

Purpose

Save current window contents in the contents buffer.

Description

Use this routine to update the contents buffer of a window without having to erase it. This is particularly useful when the wStoreContents flag is set and you are ready to save the window to a stream. Normally the window's contents are stored in the contents buffer only when the window is erased.

The window must be active or the error epNonFatal+ecWinNotActive is generated. If the wSaveContents flag is not set, SaveContents exits without doing anything.

Example

```
Win.wOptionsOn(wSaveContents);
Win.Draw;
Win.wFastWrite('New stuff', 1, 1, TextAttr);
Win.SaveContents;
```

Saves the window's current contents in the contents buffer.

See Also

Erase

ScrollHoriz / ScrollVert

Declaration

```
procedure ScrollHoriz(Cols : Integer); virtual;
```

Purpose

Scroll contents horizontally.

Description

If Cols is greater than zero, existing text in the window is shifted to the left by Cols columns and the rightmost columns are filled with the window's background character and attribute. If Cols is less than zero, existing text is shifted to the right by Cols columns and the leftmost columns are filled.

ScrollHoriz temporarily hides the mouse cursor (if a mouse is installed) to avoid mouse droppings.

Example

```
uses
  OpCrt, OpFrame, OpWindow;
var
  Win : RawWindow;
  Finished : Boolean;
  I : Byte;
begin
  Win.Init(30, 5, 50, 15);
  Win.wOptionsOn(wBordered);
  Win.Draw;
  for I := 2 to 8 do
    wFastText('Sample Text', I, 3);
  Finished := False;
  repeat
    case ReadKeyWord of
      $4B00 : Win.ScrollHoriz(-1);          {Left Arrow}
      $4D00 : Win.ScrollHoriz(1);           {Right Arrow}
      $4800 : Win.ScrollVert(-1);           {Up arrow}
      $5000 : Win.ScrollVert(1);            {Down arrow}
      $011B : Finished := True;             {Escape}
    end;
  until Finished;
  Win.Done;
end.
```

Scrolls the contents of Win in response to cursor commands. Note that once you scroll some text past an edge of the window, it's gone. You have to recreate it to bring it back.

Declaration

```
procedure ScrollVert(Rows : Integer); virtual;
```

Purpose

Scroll contents vertically.

Description

If Rows is greater than zero, existing text in the window is shifted up by Rows rows and the bottommost rows are filled with the window's background character and attribute. If Rows is less than zero, existing text is shifted down by Rows rows and the topmost rows are filled.

ScrollVert temporarily hides the mouse cursor (if a mouse is installed) to avoid mouse droppings.

Example

See ScrollHoriz.

Declaration
```
procedure SetBackChar(BackChar : Char); virtual;
```

Purpose
Set background character.

Description
Stores the specified character in a data field of the object. This character will be used when the window is cleared or scrolled.

Example
```
Win.SetBackChar(#178);
```
Sets the normal text background to the "heavy" fill character.

See Also
SetTextAttr

Declaration
```
procedure SetCursor(CT : CursorType);
```

Purpose
Set cursor shape.

Description
CT is a logical cursor shape (cuNormal, cuBlock, cuHidden, etc.) rather than physical scan lines. If the window is current at the time of the call, the actual cursor shape is changed immediately; otherwise, a data field of the object is updated and will be used when the window next becomes current.

Examples
```
Win.SetCursor(cuBlock);
```
Sets a block cursor.
```
Win.SetCursor(cuHidden);
```
Hides the cursor.

See Also
CursorType (OPCRT) SetCursor

SetFrameLimits / SetPosLimits

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure SetFrameLimits(MinW, MinH, MaxW, MaxH : Byte);
```

Purpose

Set limits for sizing of frame.

Description

MinW and MinH specify the minimum width and height of the frame; MaxW and MaxH the maximum. These limits apply to both the main and alternate frames. Attempts to resize the window beyond these limits will fail. If the frame is outside of the specified limits at the time of the call, no error is detected.

The most common use of this routine is to modify the minimum width so that a header string is guaranteed to fit in the frame.

The existing frame limits are available only by accessing the frMinW, frMinH, frMaxW, and frMaxH data fields of the window's wFrame field.

Examples

```
Win.SetFrameLimits(10, 5, ScreenWidth, ScreenHeight);
```

Sets minimum limits (for frame resizing) to 10 columns and 5 rows. Sets maximum limits to the full screen dimensions.

```
Win.SetFrameLimits(10, Win.wFrame.frMinH,
                   Win.wFrame.frMaxW, Win.wFrame.frMaxH);
```

Changes the minimum frame width to 10 columns and leaves the other limits unchanged.

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure SetPosLimits(MinX, MinY, MaxX, MaxY : Byte);
```

Purpose

Set limits for moving and drawing (generally limited by frame).

Description

The specified coordinates form the new frame-limiting rectangle (see the introduction to this section). An attempt to move or resize the window will fail if the frame–with the exception of shadows–would fall outside of this rectangle. If the frame is outside of the specified limits at the time of the call, no error is detected.

The most common use of this routine is to keep the window from moving onto some other screen item like a menu or status line.

The existing limits are available only by accessing the wMinXL, wMinYL, wMaxXH, and wMaxYH data fields of the window.

Example

```
Win.SetPosLimits(1, 2, ScreenWidth, ScreenHeight-1);
```

Protects the top and bottom rows of the screen from being overwritten by the window.

See Also

SetSizeLimits

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure SetSizeLimits(MinW, MinH, MaxW, MaxH : Byte);
```

Purpose

Set limits for sizing of active window.

Description

MinW and MinH specify the minimum width and height of the active window; MaxW and MaxH the maximum. Attempts to resize the window beyond these limits will fail. If the active window is already outside of the specified limits at the time of the call, no error is detected.

The most common use of this routine is to modify the minimum size so that particular contents will always fit within the window.

The existing window limits are available only by accessing the wMinW, wMinH, wMaxW, and wMaxH data fields of the window.

Example

```
Win.SetSizeLimits(10, 5, ScreenWidth, ScreenHeight);
```

Sets the minimum active window size to 10 columns and 5 rows. Sets the maximum to the full screen dimensions. If the window has a border, the maximum limits will probably never be critical, since the frame limits will be reached first.

See Also

SetFrameLimits SetPosLimits

Declaration

```
procedure SetTextAttr(Color, Mono : Byte); virtual;
```

Purpose

Set attributes for normal window text.

Description

Stores the specified video attributes in data fields of the object. The appropriate attribute (depending on video mode) will be used when the window is cleared or scrolled and will be assigned to OPCRT.TextAttr when the window is made current.

Example

```
Win.SetTextAttr($02, $70);
```

Normal window text will be in green on black for color monitors, reverse video for monochrome monitors.

See Also

SetBackChar

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a RawWindow in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the RawWindow to the stream S. Store first writes the fixed data fields of the RawWindow and then calls the Store methods of the component objects of the window, including Frame and ScreenRect. If the window's wStoreContents flag has been set and the window's contents rectangle is initialized at the time the stream is written, Store also writes the contents of the window (in a run-length packed format).

The stream registration procedure for a RawWindow is RawWindowStream. If the window uses explosion special effects, the stream registration procedure ExplodingWindowStream must also be called.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

Examples

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  Win : RawWindow;
  S : BufIdStream;
  WinOpts : LongInt;
  I : Word;
begin
  WinOpts := wBordered+wClear+wSaveContents+wStoreContents;
  Win.InitCustom(30, 5, 50, 15, DefaultColorSet, WinOpts);
  Win.wFrame.AddHeader('Standard Header', heTC);
  Win.wFrame.AddScrollBar(frRR, 1, 9, DefaultColorSet);
  Win.Draw;
  for I := 2 to 10 do
    Win.wFastText('Hello Mars', I, 5);
  Win.Erase;

  {Store the window to a stream}
  S.Init('RAWWIN.STM', SCreate, 1024);
  S.RegisterHier(RawWindowStream);
  S.Put(Win);
  S.Done;
  Win.Done;
end.
```

Creates a RawWindow, adds features, fills it with text, erases the window, then writes it out to a stream using the Put method, which calls RawWindow.Store indirectly. We've turned on the wStoreContents option so that Store will write out the Win.Contents object.

See BufIdStream (in OPROOT) for proper error checking using streams.

```
begin
  WinOpts := wBordered+wClear+wSaveContents+wStoreContents;
  Win.InitCustom(30, 5, 50, 15, DefaultColorSet, WinOpts);
  Win.Draw;
  for I := 2 to 10 do
    Win.wFastText('Hello Mars', I, 5);
  S.Init('RAWWIN.STM', SCreate, 1024);
```

```
    S.RegisterHier(RawWindowStream);
    S.Put(Win);
    ...
  end.
```

Probably won't produce the expected results. The wStoreContents option is on but the contents will not be written to the stream. The text in the window was never copied to the contents ScreenRect. Add a call to Win.Erase or do the following before writing the stream:

```
  Win.StoreContents;
```

See Also

Frame.Store
Load
ScreenRect.Store

TweakSlider

Declaration

```
{$IFDEF UseScrollBar}
function TweakSlider(Posn : FrameEdgeType; MousePosn : Byte;
                     UserVal : LongInt; Step : LongInt) : LongInt;
```

Purpose

Adjust user value until slider matches mouse position.

Description

Ideally, high screen resolution and the use of floating point arithmetic would provide a nearly exact correspondence between the visible slider position of a scroll bar and the associated user value. Because **Object Professional** operates in text mode and avoids the use of real arithmetic for slider mapping, these ideal assumptions are not met.

This problem becomes apparent when a user uses the mouse to point to a position on the scroll bar and clicks to scroll to that position. The RawWindow's EvaluateMousePos routine computes an equivalent user value (e.g., the line number in a text file). The program scrolls to that location and calls DrawSlider to update the slider for the new user value. However, due to roundoff errors and limited screen resolution, the slider does not coincide with the location of the mouse cursor, and the user feels unsatisfied.

TweakSlider can reduce this problem. Given a frame edge (Posn), an absolute mouse coordinate (MousePosn) that corresponds to the long direction of the scroll bar, an initial estimate for the user value (UserVal), and a step size (Step), TweakSlider returns a new user value that places the projected slider position as close as possible to the actual mouse position. TweakSlider performs an iterative search by adding or subtracting Step units to the UserVal until the best value is found.

TweakSlider performs a secondary function also aimed at matching program behavior to user expectations. It puts "english" on any scroll bar selection–for example, if the new mouse position is to the right of the existing slider on a horizontal scroll bar, TweakSlider assures that the new slider will be to the right of the existing slider, even if roundoff errors would tend to leave the slider in the same position. This function is particularly important for "chunky" scrolling like that associated with pick list columns or entry screen fields.

If MousePosn is not within the screen limits of the scroll bar, TweakSlider generates an error (epNonFatal+ecWinCoordsBad) and exits.

Example

```
repeat
  {...Wait for a mouse press...}
  EvaluateMousePos;
  UserVal := PosResults(FramePos, HotCode);
  case HotCode of
    hsNone :
      {Not on a hot spot} ;
    hsBar :
      case FramePos of
        frRR :
          {On the right hand vertical scroll bar}
          begin
            UserVal := TweakSlider(FramePos, MouseLastY+MouseYLo, UserVal, 1);
            {...Scroll window to new UserVal...}
            DrawSlider(frRR, UserVal);
          end;
        {...Other scroll bars...}
      end;
    {...Other HotCodes...}
  end;
until Finished;
```

If the mouse is clicked over the right hand scroll bar, the UserVal returned by PosResults is the relative position of the mouse cursor in the previously defined scroll bar range.

TweakSlider adjusts this UserVal so that the new slider position will better match the mouse cursor. Note that the expression MouseLastY+MouseYLo gives the absolute Y coordinate of the mouse cursor, even if a restricted mouse window is active. After computing the adjusted UserVal, we perform whatever action is needed to update the contents of the window and then redraw the slider at its new position.

For a real world example of using TweakSlider, see the QkRefChart object in the OPQKREF unit, used in DESKPOP.

See Also

EvaluatePos PosResults

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure Unzoom; virtual;
```

Purpose

Return zoomed window to pre-zoomed size.

Description

If the window is not zoomed, this method exits immediately. Otherwise it returns the window to the coordinates it had just prior to zooming.

Example

```
Win.Unzoom;
Result := Win.GetLastError;
if Result <> 0 then
  WriteLn('Failed to zoom window, Result = ', Result)
else
  {...Use unzoomed window...};
```

A zoomed window is returned to its normal size. See Zoom for a complete program showing zooming and unzooming a window.

See Also

AdjustWindow Zoom

UpdateContents

Declaration

```
procedure UpdateContents; virtual;
```

Purpose

Update the contents of the currently displayed window.

Description

Application programs should not call this method directly. RawWindow's Draw method calls it when the wUserContents flag is set and the time has come to fill in the contents of a window. RawWindow's implementation of UpdateContents does nothing. Many of the types derived from RawWindow override the method, however: PickList draws the visible section of the pick list; Menu draws all visible submenus and updates the menu item list; Memo draws the visible lines of text after accounting for vertical and horizontal scrolling; and so on.

The UpdateContents method is a good example of polymorphism in action. It accounts for the ability of **Object Professional**'s windows to share many features but to differ widely in the problems they solve.

Example

```
uses
  OpCrt, OpFrame, OpWindow;
type
  XWin = object(RawWindow)
    procedure UpdateContents; virtual;
  end;
var
  Win : XWin;

  procedure XWin.UpdateContents;
  var
    Xlo, Ylo, Xhi, Yhi, I, J : Byte;
  begin
    Coordinates(Xlo, Ylo, Xhi, Yhi);
    for I := Ylo to Yhi do
      for J := Xlo to Xhi do
        FastText(Char($40+I), I, J);
  end;

begin
  Win.Init(30, 5, 50, 15);
  Win.wOptionsOn(wBordered+wUserContents);
  Win.Draw;
  {...Use window...}
  Win.Done;
end.
```

Win is an instance of XWin (derived from RawWindow). Win overrides the UpdateContents method from RawWindow. Whenever Win.Draw is called, it will call XWin's UpdateContents to fill the window. Note that the wUserContents flag is set for this window. In a more serious example, XWin would provide its own constructor, which would set this flag whenever an instance of XWin was created.

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure UpdateScreenSize;
```

Purpose

Update window size, position, and limits for current screen size.

Description

When a RawWindow is created, default limits for resizing and repositioning the window are established. The maximum limits are set equal to the screen size in effect at the time. If the screen mode is changed thereafter, these limits will probably become invalid. In fact, the window's actual position and size could become invalid. (Suppose a window was created while in 43-line mode and then the screen was switched back to 25-line mode.) The problem becomes especially bothersome when windows can be stored and loaded from streams, since there is no guarantee that the screen size is the same when a window is loaded as when it was stored. In fact, properly supporting the streams created by MAKEMENU and MAKESCRN was the major driving force for providing this method.

UpdateScreenSize stores the value of the screen size when it was last called. If it is called again and the screen size remains unchanged, it exits immediately without wasting any time.

Otherwise, UpdateScreen updates the limits for maximum frame size, maximum active window size, and maximum position based on the new screen size. Note that it does not modify the minimum limits. If necessary, it repositions the window just enough to keep it within the new limits. If repositioning is not enough, it attempts to resize the window to keep it within limits.

The window must be either inactive or the current window when UpdateScreenSize is called. If not, it generates an error (epNonFatal+ecWinNotCurrent) and exits. If the window's size is larger than the current screen size and the wResizeable flag is not set, UpdateScreenSize generates an error (epNonFatal+ecWinCoordsBad). If the window's position or size must be adjusted, UpdateScreenSize can generate the same errors as AdjustWindow.

UpdateScreenSize works for both zoomed and unzoomed windows.

Example

```
SelectFont8x8(False);              {Assure in 25 line mode}
Win.Init(10, 5, 30, 20);           {Create and draw a window}
Win.Draw;
SelectFont8x8(True);               {Switch to 43/50 line mode}
Win.UpdateScreenSize;              {Update window limits}
Win.MoveWindow(0, +15);            {Move the window into the new screen region}
Win.Erase;
SelectFont8x8(False);              {Switch back to 25 line}
Win.UpdateScreenSize;              {Update window position and limits}
Win.Draw;                          {Redraw}
```

Creates a window while the screen is in 25 line mode and draws it. Then switches to 43 or 50 line mode (assuming an EGA or VGA) and updates the window for the new screen size. This then allows the window to be moved within the larger screen boundaries. The window is erased and the screen switched back to 25 line mode. UpdateScreenSize now must adjust the position of the window, just enough to keep it on screen. Finally the window is redrawn.

See Also

AdjustWindow

Declarations

```
procedure wFastWrite(St : string; Row, Col, Attr : Byte);
  {-Write a string}
procedure wFastText(St : string; Row, Col : Byte);
  {-Write St at Row,Col without changing the underlying video attribute}
procedure wFastVert(St : string; Row, Col, Attr : Byte);
  {-Write a string vertically}
procedure wFastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
  {-Fill Number chars at Row,Col in Attr}
procedure wChangeAttribute(Number : Word; Row, Col, Attr : Byte);
  {-Change Number video attributes to Attr starting at Row,Col}
procedure wFastCenter(St : string; Row, Attr : Byte);
  {-Write St centered on Row in Attr}
procedure wFastFlush(St : string; Row, Attr : Byte);
  {-Write St flush right on Row in Attr}
procedure wFastRead(Number, Row, Col : Byte; var St : string);
  {-Read Number characters from the screen into St starting at Row,Col}
procedure wReadAttribute(Number, Row, Col : Byte; var St : string);
  {-Read Number attributes from the screen into St starting at Row,Col}
procedure wWriteAttribute(St : String; Row, Col : Byte);
  {-Write string of attributes St at Row,Col without changing characters}
procedure wFlexWrite(St : string; Row, Col : Byte; var FAttrs : FlexAttrs);
  {-Write a string flexibly using window-relative coordinates}
procedure wFastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
  {-Write St at Row,Col in Attr (video attribute) without snow.
    Control characters displayed in Ctrl as upper-case letters}
```

Purpose

Write using coordinates relative to the active window boundary.

Description

These methods provide a number of convenient ways to write to or read from the contents of a window. If the frame has a border, the (1, 1) position for these routines is just inside the upper left hand corner of the border. If the frame has no border, the (1, 1) position is coincident with the upper left corner of the frame.

Note that the coordinates passed to each routine are in Row,Col order. The valid Row range is 1 to Win.Height and the Col range is 1 to Win.Width. These routines clip at the window boundaries when appropriate. They will not allow you to write or read outside of the active window.

If the window is not active at the time of the call, each routine generates an error (epNonFatal + ecWinNotActive) and exits. Note that the window need not be current in order to write to it. It is your responsibility to guarantee that the window is not obscured by another window, however; these routines write blindly to the specified coordinates. If you need to write to partially obscured windows, define each window as a StackWindow and use the methods of the same name from that object.

wFastWrite writes in a specified video attribute. The application must pick the attribute appropriately for color and monochrome video modes.

wFastText writes the string without changing the underlying video attribute. Note that this does not necessarily produce the same result as calling wFastWrite with an attribute of TextAttr.

wFastVert writes its string vertically.

wFastFill fills the window with Number pairs of character Ch and attribute Attr. If Number is greater than the window's width, the fill continues on the next row of the window.

wChangeAttribute changes Number video attributes to the specified value without changing the existing characters. Like wFastFill, it wraps at the window boundaries.

wFastCenter writes the string centered in the window's width on the specified Row in the specified attribute.

wFastFlush writes the string justified against the right edge of the window.

wFastRead reads Number characters from the window into the St variable. Be sure that the variable you pass for St is long enough to hold at least Number characters! If the type of your string is anything other than the generic String type, you'll need to use the {$V-} compiler option to relax string type checking. If Number is larger than the number of characters on the specified Row, only the remaining characters on the row are returned.

wReadAttribute is like wFastRead except that it reads the video attributes rather than the characters.

wWriteAttribute changes the attributes to the values specified in St, without changing the existing characters.

wFlexWrite writes a string in one or more video attributes. See OPCRT.FlexWrite for more information.

wFastWriteCtrl is like wFastWrite except that it treats control characters (those with ordinal less than 32) specially. Control characters are mapped to the equivalent upper case alphabetic and are then written using the Ctrl attribute. For example, Ctrl-A (ordinal 1) would be written as the letter A.

Examples

```
var S : String;
...
Win.wFastWrite('Hello World', 1, 1, TextAttr);
Win.wFastRead(5, 1, 1, S);
```

Writes 'Hello World' to the upper left corner of the window in the default text attribute for the current window, then fills S with the first five characters starting at the same position (S will contain 'Hello').

```
Win.Init(30, 5, 50, 15);
Win.wFastFill(Win.Width*Win.Height, #178, 1, 1, TextAttr);
```

Fills the entire window with the "heavy" background character.

```
Win.Init(30, 5, 50, 15);
Win.wFastFill(300, #178, 1, 1, TextAttr);
```

The arithmetic is wrong (there's only 231 bytes in this window), but wFastFill will clip the extra characters.

```
Win.Init(30, 5, 50, 15);
Win.wFastWrite('Hello', 12, 1, $02);
```

Incorrect! The maximum row in this window is 11. However, wFastWrite will catch this error and exit without doing anything (no result code is set).

```
const
  FAttrs : FlexAttrs = ($07, $01, $02, $03);
...
Win.Init(30, 5, 50, 15);
Win.wFlexWrite('This '^A'is'^A' '^B'a'^B' '^C'test', 1, 1, FAttrs);
```

Each of the four words in the sentence 'This is a test' is displayed in a different attribute.

See the related procedures in OPCRT for more examples.

See Also

Frame-relative Writing

wCopy / wGotoXY

Declaration

```
constructor wCopy(var W : RawWindow);
```

Purpose

Initialize a window from another one, making a unique copy of all dynamic structures.

Description

After making a copy with wCopy, you can modify the copy without affecting the original window. If the contents buffer of the original window was initialized, it will also be initialized for the copy.

Example

```
var
  Win1, Win2 : RawWindow;
begin
  Win1.InitCustom(2, 2, 10, 10, DefaultColorSet,
                  DefWindowOptions or wBordered);
  Win1.wFrame.AddHeader(' First Window ', heTL);
  if not Win2.wCopy(Win1) then begin
    Writeln('Failed to Init Win2, Status ', InitStatus);
    Halt;
  end;
  Win1.Done;
  Win2.Draw;
end.
```

Win1 is instantiated by InitCustom; Win2 is instantiated by wCopy and is an exact duplicate of Win1, including its frames and other components.

Declaration

```
procedure wGotoXY(X, Y : Integer); virtual;
```

Purpose

Move cursor to window-relative coordinates.

Description

For the current window, wGotoXY works just like the GotoXY procedure in the OPCRT unit by repositioning the hardware cursor to window-relative coordinates X and Y. wGotoXY also works for non-current windows: it stores the specified position in data fields of the window; they will take effect the next time the window is made current.

There is one other difference from GotoXY: wGotoXY allows the cursor to be positioned outside of the window. Even if X or Y is zero, negative, or larger than the window size, wGotoXY will still position the cursor as requested. wGotoXY neither detects nor rejects requests to move the cursor off the screen.

Example

```
Win.Init(30, 5, 50, 15);
Win.Draw;
Win.wGotoXY(5, 5);
Write('X');
```

Draws an 'X' five columns over and five rows down from the top-left corner of the window.

See Also

wWhereXY wWhereXYAbs

Declaration
```
function Width : Word;
```

Purpose
Return the active width of the window.

Description
Returns the number of columns in the active window rectangle.

This method is actually implemented in the AbstractWindow object.

Example
```
Win.wFrame.AddCustomShadow(#178, frBB, 1, 1, Win.Width+2, 1, $02, $70);
Win.wFrame.AddCustomShadow(#178, frRR, 1, 1, 1, Win.Height+2, $02, $70);
```

Uses the Width function to add a shadow the same width as the frame (2 columns larger than the window).

See Also
Height

Declaration
```
function wOptionsAreOn(OptionCodes : LongInt) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
OptionCodes is a bitmapped long integer using the bit masks described in the Declarations area of this section. wOptionsAreOn returns True only if *all* of the specified options are set.

Multiple options are combined by numerically or'ing them.

Examples
```
if Win.wOptionsAreOn(wBordered) then begin
  Win.wFrame.AddHeader('Standard Header', heTC);
  Win.wFrame.AddShadow(shBR, shOverWrite);
end;
```

Checks to see if the window has a border before it adds frame features.

```
if Win.wOptionsAreOn(wSaveContents or wUserContents) then
  ...
```

Probably incorrect. These two options are rarely set at the same time, so the expression won't return True. What is probably meant is the following:

```
if Win.wOptionsAreOn(wSaveContents) or
   Win.wOptionsAreOn(wUserContents) then
  {The window can restore its contents}
  ...
```

See Also
wOptionsOff wOptionsOn

wOptionsOff

Declaration

```
procedure wOptionsOff(OptionCodes : LongInt);
```

Purpose

Turn options off.

Description

The options specified by OptionCodes are all turned off. Multiple options are combined by numerically or'ing them.

This method may be called only when the window is inactive. Otherwise it generates an error (epNonFatal+ecWinIsActive) and exits immediately. wOptionsOff may generate other errors under rare circumstances.

If the wBordered option is turned off, the frames' coordinates are pulled in by one on all edges; any frame features (headers, shadows, etc.) are removed and disposed. Window contents stored in the contents buffer are also invalidated.

Example

```
Win.InitCustom(30, 5, 50, 15, DefaultColors, DefWindowOptions or wClear);
{...}
Win.wOptionsOff(wClear);
```

The window is instantiated with the wClear option set; later in the program that option is turned off.

See Also

wOptionsOn

Declaration

```
procedure wOptionsOn(OptionCodes : LongInt);
```

Purpose

Turn options on.

Description

The options specified by OptionCodes are all turned on. Multiple options are combined by numerically or'ing them. (Adding the options together is equivalent only if the specified options do not contain overlapping bits.)

This method may be called only when the window is inactive. Otherwise it generates an error (epNonFatal + ecWinIsActive) and exits immediately. wOptionsOn may generate other errors such as (epFatal + ecOutOfMemory) or (epFatal + ecWinCoordsBad).

If the wBordered option is turned on, the frames' coordinates are expanded by one on all edges; any existing frame features (headers, shadows, etc.) are removed and disposed. Window contents stored in the contents buffer are also invalidated. The window gets the default border type (DefWindowFrame).

Note that it is more efficient to created bordered windows by using the InitCustom constructor and setting the wBordered option during the call to the constructor. Using the wOptionsOn method to add a frame guarantees heap fragmentation: the ScreenRect buffer for the region covered by the window must be deallocated and a slightly larger buffer reallocated when the border is added.

Examples

```
Win.Init(30, 5, 50, 15);
Win.wOptionsOn(wBordered);
if Win.GetLastError <> 0 then
  WriteLn('Failed to add border');
```

This window is instantiated without a border (default window options don't normally include a border). wOptionsOn turns on the border. Checking for errors is important here since activating the border may invalidate the window coordinates (if they can't be expanded by one) and may exhaust the heap (since the covers buffer must grow).

```
Win.Init(30, 5, 50, 15);
Win.Draw;
Win.wOptionsOn(wResizeable);
{...Use window...}
Win.Done;
```

Incorrect! You can't turn options on (or off) while a window is active.

```
Win.wOptionsOn(wResizeable);
```

This example does not require any error checking (assuming the window is not active at the time of the call) since it simply sets an option bit. The following options are in the same category: wUserContents, wSoundEffects, wAltFrame, wSetMouse, wFullMouseWindow, wResizeable, wAllMouseEvents, and wStoreContents.

See Also

wOptionsAreOn wOptionsOff

wWhereXY / wWhereXYAbs

Declaration

```
procedure wWhereXY(var X, Y : Integer); virtual;
```

Purpose

Return window-relative coordinates of cursor.

Description

If the window is current, wWhereXY returns the position of the hardware cursor in window-relative coordinates. Otherwise, it returns the position the cursor will have when the window is made current. Note that X and Y may be zero, negative, or larger than the window size. If so, the cursor is outside of the active window.

Example

```
var
  X, Y : Integer;
...
Win.Init(30, 5, 50, 15);
Win.Draw;
Win.wGotoXY(1, 1);
Win.wWhereXY(X, Y);
```

After the call to wWhereXY, X and Y will both be equal to one.

Declaration

```
procedure wWhereXYAbs(var X, Y : Byte);
```

Purpose

Return absolute coordinates of cursor.

Description

If the window is current, wWhereXYAbs returns the position of the hardware cursor in absolute coordinates. Otherwise, it returns the position the cursor will have when the window is made current.

Example

```
Win.Init(30, 5, 50, 15);
Win.Draw;
Win.wGotoXY(1, 1);
Win.WhereXYAbs(X, Y);
```

After the call to WhereXYAbs, X will equal 30, Y will equal 5.

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure Zoom; virtual;
```

Purpose

Increase window size to maximum limits temporarily.

Description

Zoom saves the current window coordinates (to be restored by a subsequent call to Unzoom) and calls AdjustWindow to increase the window to its maximum size. In doing so, it sets the frame coordinates equal to those of the frame limiting rectangle. All related frame structures, such as headers and shadows, are adjusted automatically.

To zoom a window, it must either be inactive or the current window. If the window is current, the change shows up immediately. Zoom uses virtual screen techniques to make an apparently instant transition from one size to another. Note that the virtual screen requires a temporary heap buffer of 2*ScreenWidth*ScreenHeight bytes.

Zoom can generate any of a number of errors and warnings, including:

```
epWarning+ecWinIsZoomed      if the window is already zoomed
epNonFatal+ecWinNotSizeable  if the window's wResizeable flag is not set
epNonFatal+ecWinNotCurrent   if the window is not either current or inactive
epNonFatal+ecWinCoordsBad    if the various limiting dimensions of the
                             window are not self-consistent
epFatal+ecOutOfMemory        if insufficient heap space is available to
                             increase buffer sizes
```

Note that calling Zoom is guaranteed to fragment the heap (because the covers buffer must be deallocated and reallocated with a larger size). Writing an application that runs continuously for a long period of time with random sequences of window zooming, unzooming, and resizing can exhaust heap resources, leading to an unrecoverable error. Unfortunately, given these circumstances, there is no easy way to avoid this problem.

Example

```
uses OpCrt, OpFrame, OpWindow;
var Win : RawWindow;
    Result : Word;
begin
  with Win do begin
    Init(30, 5, 50, 15);
    wOptionsOn(wBordered);
    wFrame.AddScrollBar(frRR, 0, 100, DefaultColorSet);
    Draw;
    Zoom;
    Result := GetLastError;
    if Result <> 0 then
      WriteLn('Failed to zoom window, Result = ', Result);
    Unzoom;
    Result := GetLastError;
    if Result <> 0 then
      WriteLn('Failed to unzoom window, Result = ', Result);
    Done;
  end;
end.
```

Zooms the window immediately after drawing it; later unzooms it.

See Also

AdjustWindow
ResizeWindow
Unzoom

2. StackWindow

Whenever a StackWindow is drawn, a pointer to it is pushed onto a stack. By referring to these pointers, the StackWindow methods can act more intelligently than the same routines in a RawWindow. The major advantages are the ability to write to a window that is partially obscured by other windows and to select such a window and pull it to the top of the stack.

The added storage required to implement these capabilities is minimal. OPWINDOW defines another object--the WindowStack, which is derived from a PointerStack--that works closely with the StackWindow to manage window relationships. The default WindowStack can manage up to 8 windows at a time, using only 32 bytes of heap space. The WindowStack will dynamically expand itself if more than 8 windows are active at once.

Each StackWindow instance uses 10 bytes more storage than does a RawWindow. Four of these bytes form a pointer to the WindowStack where the window was pushed when it was drawn. The cross-coupling between these two objects results in better speed and also opens the possibility of managing multiple window stacks simultaneously. Another four bytes are used to point to a user-defined procedure which is called when the window is pushed onto the stack. This procedure, in combination with a word-size window number assigned by the WindowStack, can update a specialized header containing each window's number.

StackWindow and WindowStack depend on the VirtScreen object to avoid unsightly screen flash while manipulating the window stack. Suppose you have four overlapping windows on screen and want to select the bottommost one and pull it to the front. The WindowStack manager erases all four windows, redraws three of them, and finally redraws the desired one at the top of the stack. If these individual steps were visible--no matter how fast they occurred--the visual effect would be extremely ugly. OPWINDOW instead draws updates on a virtual screen which is only copied to the physical one after the operation is complete, making the transition look clean and instantaneous.

The virtual screen allocates enough heap space to hold the contents of the physical screen. Keep in mind that at least 2*ScreenWidth*ScreenHeight bytes of heap space must be free when you call almost any method of a hidden StackWindow.

In order to perform these temporary unstacking and restacking operations, the window manager must enforce an additional requirement: Each window must be able to restore its own contents when it is redrawn shortly after being erased. In the OPWINDOW hierarchy, this means that either the wSaveContents flag is set and a contents buffer has been allocated, or that the wUserContents flag is set and the window has an UpdateContents method that can regenerate the contents on demand. Every StackWindow should select one of these options or the window manager is likely to produce the error epNonFatal+ecWinInaccessible. All of **Object Professional**'s higher level types such as TextEditor, PickList, and Menu meet this requirement.

StackWindow overrides many RawWindow methods. We won't need to repeat all of their descriptions; they act just like those in RawWindow except that they work on windows that are active but not current. (Calling the RawWindow versions of the methods would corrupt any overlapping window.) StackWindow overrides the following methods for this purpose:

Clearing, Scrolling, and Saving

```
procedure Clear; virtual;
  {-Clear the window to wTextXxx and wBackChar}

procedure ClearContents;
  {-Clear region within active frame with wTextXxx and wBackChar}
```

```
  procedure ScrollHoriz(Cols : Integer); virtual;
    {-Scroll contents horizontally}
  procedure ScrollVert(Rows : Integer); virtual;
    {-Scroll contents vertically}
  procedure SaveContents; virtual;
    {-Save current window contents in the contents buffer}
```

Moving and Resizing

```
  procedure AdjustWindow(X1, Y1, X2, Y2 : Word); virtual;
    {-Set new coordinates and adjust all related structures}
  procedure MoveWindow(DX, DY : Integer); virtual;
    {-Move a window}
  procedure ResizeWindow(DX, DY : Integer); virtual;
    {-Resize a window}
  procedure Zoom; virtual;
    {-Increase window size to specified limits temporarily}
  procedure Unzoom; virtual;
    {-Return zoomed window to original size}
```

Scroll Bars

```
  procedure DrawAllSliders(HorizVal, VertVal : LongInt); virtual;
    {-Draw or update all sliders}
  procedure ChangeAllScrollBars(MinHoriz, MaxHoriz : LongInt;
                                MinVert, MaxVert : LongInt); virtual;
    {-Change user ranges of existing scroll bars. Erases sliders}
```

Window-relative Writing

```
  procedure wFastWrite(St : string; Row, Col, Attr : Byte);
    {-Write a string using window-relative coordinates}
  procedure wFastText(St : string; Row, Col : Byte);
    {-Write St at Row,Col without changing the underlying video attribute.}
  procedure wFastVert(St : string; Row, Col, Attr : Byte);
    {-Write a string vertically using window-relative coordinates}
  procedure wFastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
    {-Fill Number chs at Row,Col in Attr without snow}
  procedure wFastCenter(St : string; Row, Attr : Byte);
    {-Write St centered on Row in Attr without snow}
  procedure wFastFlush(St : string; Row, Attr : Byte);
    {-Write St flush right on Row in Attr without snow}
  procedure wFastRead(Number, Row, Col : Byte; var St : string);
    {-Read Number characters from the screen into St starting at Row,Col}
  procedure wReadAttribute(Number, Row, Col : Byte; var St : string);
    {-Read Number attributes from the screen into St starting at Row,Col}
  procedure wWriteAttribute(St : String; Row, Col : Byte);
    {-Write string of attributes St at Row,Col without changing characters}
  procedure wChangeAttribute(Number : Word; Row, Col, Attr : Byte);
    {-Change Number video attributes to Attr starting at Row,Col}
  procedure wFlexWrite(St : string; Row, Col : Byte; var FAttrs : FlexAttrs);
    {-Write a string flexibly using window-relative coordinates}
  procedure wFastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
    {-Write St at Row,Col in Attr (video attribute) without snow.
      Control characters displayed in Ctrl as upper-case letters}
```

Frame-Relative Writing

```
procedure fFastWrite(St : string; Row, Col, Attr : Byte);
  {-Write a string}
procedure fFastText(St : string; Row, Col : Byte);
  {-Write St at Row,Col without changing the underlying video attribute}
procedure fFastVert(St : string; Row, Col, Attr : Byte);
  {-Write a string vertically}
procedure fFastFill(Number : Word; Ch : Char; Row, Col, Attr : Byte);
  {-Fill Number chars at Row,Col in Attr}
procedure fChangeAttribute(Number : Word; Row, Col, Attr : Byte);
  {-Change Number video attributes to Attr starting at Row,Col}
procedure fFastCenter(St : string; Row, Attr : Byte);
  {-Write St centered on Row in Attr}
procedure fFastFlush(St : string; Row, Attr : Byte);
  {-Write St flush right on Row in Attr}
procedure fFastRead(Number, Row, Col : Byte; var St : string);
  {-Read Number characters from the screen into St starting at Row,Col}
procedure fReadAttribute(Number, Row, Col : Byte; var St : string);
  {-Read Number attributes from the screen into St starting at Row,Col}
procedure fWriteAttribute(St : String; Row, Col : Byte);
  {-Write string of attributes St at Row,Col without changing characters}
procedure fFlexWrite(St : string; Row, Col : Byte; var FAttrs : FlexAttrs);
  {-Write a string flexibly using window-relative coordinates}
procedure fFastWriteCtrl(St : String; Row, Col, Attr, Ctrl : Byte);
  {-Write St at Row,Col in Attr (video attribute) without snow.
    Control characters displayed in Ctrl as upper-case letters}
```

The typical implementation of these routines looks as follows:

```
procedure StackWindow.Clear;
  {-Clear the window to wTextXxx and wBackChar}
var NotC : Boolean;
begin
  NotC := not IsCurrent;
  if NotC then begin
    ActivateWrite;
    if wRes <> 0 then Exit;
  end;
  RawWindow.Clear;
  if NotC then
    DeactivateWrite;
end;
```

If the window is already the current window, StackWindow.Clear simply calls RawWindow.Clear. If the window is not current (or in this case if it's not even active), it calls ActivateWrite. ActivateWrite first searches from the top of the stack to find the desired window. If it fails, or if the search encounters a window without its wSaveContents or wUserContents flag set, ActivateWrite will fail with the error epNonFatal+ecWinInaccessible. Next ActivateWrite allocates a heap buffer for the virtual screen, copies the physical screen to the virtual one, and redirects the video output of OPCRT and OPWINDOW to the virtual screen. If there is insufficient heap space, it generates the error epFatal+ecOutOfMemory. Finally, ActivateWrite erases the windows that were pushed onto the stack after the specified one. (This step occurs on the virtual screen, so it isn't visible.)

If no error has occurred, it's now safe to call RawWindow.Clear since the desired window has become the current one. The Clear operation occurs on the virtual screen.

If the window was not current to begin with, DeactivateWrite reverses the process by redrawing the temporarily erased windows on the virtual screen, redirecting video output to the physical screen, copying the virtual screen to the physical one, and disposing of the heap buffer.

Of course, all of this happens in an instant; the hidden window seems to clear itself magically without disturbing the windows that cover it.

It's worth keeping ActivateWrite and DeactivateWrite in mind. By using these in combination with other methods of the higher level window objects, it's possible to perform almost any kind of screen update on a hidden window. If you run the MAKEMENU and MAKESCRN utility programs you'll notice many such examples.

Before we proceed with detailed descriptions of the StackWindow types and methods, let's present a complete example showing how to use the StackWindow.

```
program StackWindowExample;  {EXSWIN.PAS}
uses
  OpString, OpCrt, OpFrame, OpWindow;
var
  Win : array[1..3] of StackWindowPtr;
  WinOpts : LongInt;
  I, J : Integer;
begin
  {Make three bordered StackWindows}
  WinOpts := wBordered+wClear+wSaveContents;
  New(Win[1], InitCustom(30, 03, 50, 13, DefaultColorSet, WinOpts));
  New(Win[2], InitCustom(35, 08, 55, 18, DefaultColorSet, WinOpts));
  New(Win[3], InitCustom(40, 13, 60, 23, DefaultColorSet, WinOpts));
  if (Win[1] = nil) or (Win[2] = nil) or (Win[3] = nil) then begin
    WriteLn('Failed to initialize StackWindows');
    Halt;
  end;

  {Add some frame features and draw each window}
  for I := 1 to 3 do
    with Win[I]^ do begin
      wFrame.AddHeader('Window '+Long2Str(I), heTC);
      wFrame.AddShadow(shBR, shOverWrite);
      wFrame.AddScrollBar(frRR, 1, 9, DefaultColorSet);
      EnableExplosions(20);
      Draw;
      for J := 2 to 10 do
        wFastText('Hello Mars', J, 5);
      DrawSlider(frRR, I);
    end;

  {Switch between the windows until <Escape> is pressed}
  I := 1;
  while ReadKeyWord <> $011B do begin
    Win[I]^.Select;
    Inc(I);
    if I > 3 then
      I := 1;
  end;
```

```
    {Clear window 1, even it it is hidden}
    Win[1]^.Clear;

    {Erase and dispose of each window}
    for I := 1 to 3 do begin
      J := ReadKeyWord;
      Win[I]^.EraseHidden;
      Dispose(Win[I], Done);
    end;

    {Deallocate the window stack}
    wStack.Done;
  end.
```

The example declares an array of three pointers to StackWindows. (It could just as easily declare an array of StackWindows without using the pointers.) Each of the three StackWindow objects is allocated using a New statement which also calls the InitCustom constructor to assign the window coordinates, colors, and options.

The first For loop adds frame features to each window (using capabilities inherited from RawWindow and Frame), then draws it and writes some text to it. The While loop demonstrates random access to the windows, pulling each one to the front in turn, until the <Esc> key is pressed. After selecting a window it is the current window, at the top of the stack and fully visible.

Next we show that the Clear method will clear any of the active windows, hidden or not. EraseHidden erases any window and Done disposes of it. (Note that StackWindow.Erase will erase a window only if it is current. This decision was made so that you could avoid linking in the virtual screen code if you didn't need it.)

The final call to wStack.Done is optional, but worth explaining here. wStack is the global WindowStack variable that manages the active windows. As mentioned before, wStack has a 32 byte buffer to keep track of up to 8 windows (or a bigger buffer if more windows have been activated). By calling wStack.Done, we free up all of the heap space the program has allocated.

Mixing RawWindow and StackWindow Objects

It may sometimes be appropriate to have a mixture of RawWindow and StackWindow objects on the screen at the same time. Doing so requires additional care since the WindowStack manager has no knowledge of the RawWindows, and the RawWindows only distinguish between the CurrentWindow and not-the-current-window.

If a RawWindow is drawn prior to any StackWindow, and all the StackWindows are erased before erasing the RawWindow, there is no difficulty. Just use the natural calls in this case. There is also no problem in writing to the RawWindow (using either the FastWrite-family procedures or the object's wFastWrite methods) at any time, assuming that the RawWindow is not obscured by any other window.

More care is required when a RawWindow must be drawn after StackWindows are drawn and erased while StackWindows are still active. Drawing the RawWindow is no problem: just call its Draw method, which makes it the current window. To transfer control to any StackWindow, use the StackWindow.Select method, which includes special code that effectively builds a bridge between the two types. To transfer control back to the RawWindow, call its Draw method again. It is especially important to Select a StackWindow immediately after erasing the RawWindow, since the RawWindow doesn't know which window to make current after it is erased.

Types

```
StackWindowProc   = procedure (WPtr : WindowPtr);
```

StackWindow's Draw and DrawHidden methods call a procedure of this type just before actually drawing the specified window. Before calling the procedure, the window is assigned a unique small integer number, which may be determined by calling the WindowNumber method. If appropriate, the StackWindowProc can use the opportunity to update a header which will display the window's number. DESKPOP takes advantage of this facility to identify each of its desktop windows. Assign a StackWindowProc by calling the SetStackProc method.

```
WindowPtr         = ^StackWindow;
StackWindowPtr    = ^StackWindow;
StackWindow       =
  object(RawWindow)
    ...
  end;
```

Data types describing the StackWindow object. Note that WindowPtr is a synonym for StackWindowPtr. This shortened name is made available both for historical reasons (the StackWindow is the closest match to Turbo Professional's window type, which was referred to via a WindowPtr) and also because the StackWindow is the most useful general window type.

ActivateWrite

Declaration

```
procedure ActivateWrite;
```

Purpose

Activate fastwriting to a hidden window.

Description

In order for ActivateWrite to succeed, the window must be active, any windows drawn on top of it must have either the wSaveContents or the wUserContents flag set, and sufficient heap space must exist to activate a virtual screen the same size as the physical screen (2*ScreenWidth*ScreenHeight bytes). Failing these will result in the error epNonFatal + ecWinInaccessible or epFatal + ecOutOfMemory.

Assuming these prerequisites hold, ActivateWrite allocates the virtual screen, copies the physical screen to it, redirects video output to the virtual screen, and erases any windows stacked on top of the desired one.

Thereafter, any "fastwrite-based" procedure or method for writing to the screen will go to the proper position on the virtual screen. About the only procedures you *cannot* call successfully at this point are Write and WriteLn. You *can* call OPCRT.FastWrite and related routines; you *can* call wFastWrite and related window-relative methods; you *can* call fFastWrite and related frame-relative methods; you *can* call any higher level methods that are built upon these lower level routines.

Changes made will not be visible until DeactivateWrite is called. You must call DeactivateWrite before switching to or drawing another window.

If you have more than one update to make to a hidden window, it is faster to use ActivateWrite and DeactivateWrite than to call a method such as StackWindow.wFastWrite for each update. Each call to wFastWrite would generate internal calls to ActivateWrite and DeactivateWrite anyway.

Note that you can call ActivateWrite even for the current window. Screen updates are still temporarily redirected to a virtual screen, thereby allowing you to perform multiple updates which will appear simultaneously when DeactivateWrite is called.

Examples

```
Win[3]^.Select;
Win[1]^.ActivateWrite;
Win[1]^.Clear;
Win[1]^.wFastWrite('FastWrite', 1, 1, TextAttr);
Win[1]^.wFastCenter('FastCenter', 2, TextAttr);
Win[1]^.DeeactivateWrite;
```

Since Win[1] isn't current, we must surround the output statements with ActivateWrite and DeactivateWrite. We could even call the non-object-oriented procedure FastWrite here, although its coordinates are not window-relative. Note that it would be more efficient to use a With statement (with Win[1]^ do begin) here; we've avoided that to make the example clearer.

```
Win[3]^.Select;
Win[1]^.ActivateWrite;
Win[1]^.wFastWrite('FastWrite', 1, 3, TextAttr);
Win[2]^.Draw;
```

Incorrect! This example doesn't have a call to DeactivateWrite. The screen is not updated until DeactivateWrite is called.

See Also

DeactivateWrite

Declaration
```
procedure AssignNumber; virtual;
```

Purpose
Called to assign window number.

Description
The Draw and DrawHidden methods call this virtual method just prior to pushing the window on the stack, which occurs prior to actually drawing the window. By default, AssignNumber calls a WindowStack method which returns the next free window number. The first window number is 1, the second 2, and so on. If a window was previously assigned a window number and then erased, its window number will be reused. For example, if windows 1, 2, and 3 were drawn and window 2 was erased, then the next window drawn would get the number 2. (The highest window number is 2047. If there are more windows active than this, they will be assigned number 0.)

AssignNumber assigns this window number to a data field of the object and then calls the window's stack proc (see the StackWindowProc type above). By assigning your own procedure for the stack proc, you can arrange to display the window number.

Objects derived from StackWindow can get different behavior by overriding the AssignNumber method.

See Also
SetStackProc WindowNumber

Declaration
```
procedure DeactivateWrite;
```

Purpose
Deactivate fastwriting and display changes.

Description
Reverses the effect of ActivateWrite. See that routine for more information.

See Also
ActivateWrite

Draw / DrawHidden

Declaration

```
procedure Draw; virtual;
```

Purpose

Push window on stack and draw it.

Description

If the window is already the current window, Draw just calls UpdateContents and exits. If the window is active but not current, Draw generates an error (epNonFatal+ecWinInaccessible) and exits. To bring an active window to the top of the window stack you should call the Select method instead. To write to a hidden window, call ActivateWrite or use one of the methods mentioned in this section's introduction.

If the window is not active, Draw calls the AssignNumber method, pushes the window's address onto the global WindowStack variable, wStack, and finally calls RawWindow.Draw to display the window. This window then becomes current.

Example

```
uses
  OpCrt, OpWindow;
var
  Wins : array[1..3] of StackWindow;
  WinOpts : LongInt;
  I : Integer;
begin
  {Initialize three windows}
  WinOpts := wBordered+wClear+wSaveContents;
  Wins[1].InitCustom(30, 03, 50, 13, DefaultColorSet, WinOpts);
  Wins[2].InitCustom(35, 08, 55, 18, DefaultColorSet, WinOpts);
  Wins[3].InitCustom(40, 13, 60, 23, DefaultColorSet, WinOpts);

  {Draw each window}
  for I := 1 to 3 do
    Wins[I].Draw;
end.
```

Declares an array of three static StackWindow variables, initializes them, then draws them.

See Also

AssignNumber Select

Declaration

```
procedure DrawHidden(var Under : StackWindow);
```

Purpose

Draw a window, inserting it in the stack underneath window Under.

Description

The window must not already be active or DrawHidden will generate the error epNonFatal+ecWinIsActive and exit. If the window Under is not active, DrawHidden just calls StackWindow.Draw, placing the new window on top of the stack. Otherwise DrawHidden assures that the windows down to and including Under can restore themselves (see ActivateWrite). If not, it generates epNonFatal+ecWinInaccessible and exits. If so, it creates and activates a virtual screen, erases all the windows down to and including Under, pushes and draws the new window, redraws the windows just removed, and finally deactivates the virtual screen to make the changes visible.

Example

```
Win[3]^.DrawHidden(Win[2]^);
```

Draws Win[3] underneath Win[2] on the stack, assuming that Win[2] was already visible.

Declaration
```
procedure Erase; virtual;
```

Purpose
Erase a window and pop from stack.

Description
The window to be erased must be the top window on the stack. If the window is not active, Erase generates the epNonFatal+ecWinNotActive error; if the window is not current, it generates epNonFatal+ecWinInaccessible. To erase an active, non-current window, call EraseHidden.

When the window is erased, it is assigned a window number of zero (see StackWindow.Draw). The window remaining at the top of the stack, if any, is updated to assure that it displays its main frame (wFrame) instead of the alternate frame.

Examples
```
for I := 1 to 3 do
  Win[I]^.Draw;
{...Use Win[3]...}
Win[3]^.Erase;
```

Draws three windows, works with the third window, then erases the third window.

```
for I := 1 to 3 do
  Win[I]^.Draw;
{...Use Win[3]...}
Win[1]^.Erase;
```

Incorrect! Only the top window on the stack can be erased. This example will return a "window inaccessible" error. Use EraseHidden instead or Select the window first.

See Also
WindowStack.TopWindow

Declaration
```
procedure EraseHidden;
```

Purpose
Erase a window even if it's hidden on the window stack.

Description
Works like Erase except that it uses virtual screen techniques so that any window on the stack can be erased. Erase itself does not implement this behavior in order to avoid linking the virtual screen code for applications that don't otherwise require it.

Example
```
WinOpts := wBordered+wClear+wSaveContents;
New(Win[1], InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
New(Win[2], InitCustom(35, 8, 55, 18, DefaultColorSet, WinOpts));
for I := 1 to 2 do
  Win[I]^.Draw;
{...Use Win[2]...}
Win[1]^.EraseHidden;
```

After drawing two windows (leaving Win[2] as the current window) we decide to erase Win[1]. Since Win[1] is not current, we use EraseHidden.

Init / InitCustom

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte);
```

Purpose

Create a stacked window with default options and colors.

Description

Generates a call to StackWindow.InitCustom, passing DefaultColorSet in the Colors parameter and DefWindowOptions in Options.

Example

```
var Win : array[1..3] of WindowPtr;
begin
  New(Win[1], Init(30, 3, 50, 13));
  New(Win[2], Init(35, 8, 55, 18));
  New(Win[3], Init(40, 13, 60, 23));
  if (Win[1] = nil) or (Win[2] = nil) or (Win[3] = nil) then begin
    WriteLn('Failed to Init StackWindow'); Halt;
  end;
end;
```

Instantiates three StackWindows with default colors and options. Note that the wSaveContents flag is set in the default window options, making these windows compatible with the "random access" StackWindow methods like ActivateWrite, DrawHidden, Select, and EraseHidden.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet; Options : LongInt);
```

Purpose

Create a window with custom options.

Description

InitCustom first calls RawWindow.InitCustom to perform the bulk of the initialization. Then it points the window's associated stack to wStack, the default global WindowStack. It also assigns a window number of zero and points the window's StackWindowProc to a routine that does nothing.

If the constructor fails, an error code will be stored in InitStatus.

Example

```
var
  Win : array[1..2] of WindowPtr;
  WinOpts : LongInt;
begin
  WinOpts := wBordered+wClear+wSaveContents+wResizeable;
  New(Win[1], InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
  New(Win[2], InitCustom(35, 8, 55, 18, DefaultColorSet, WinOpts));
  if (Win[1] = nil) or (Win[2] = nil) then begin
    WriteLn('Failed to Init StackWindow, Status ', InitStatus);
    Halt;
  end;
end.
```

Creates two StackWindows with default colors but with custom options (windows will be bordered, resizeable, cleared when first drawn, and have their contents saved when erased).

See Also

RawWindow.InitCustom

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load StackWindow from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to StackWindow.Store. Load first calls RawWindow.Load to read its ancestor's data. Then it reads the window number and the pointer code for the window's stack proc.

If the stack proc's address was registered before Store was called, and it is registered again prior to the call to Load, then Load will initialize the StackWindow to use the procedure whose address was registered. Otherwise, Load assigns the default stack proc (which does nothing) to the window.

The stream registration procedure for a StackWindow is StackWindowStream.

Example

```
uses
  OpString, OpCrt, OpRoot, OpFrame, OpWindow;
var
  Win : array[1..2] of WindowPtr;
  S : BufIdStream;
  I : Word;
{$F+}
procedure StackProc(WPtr : WindowPtr);
begin
  with WPtr^ do
    ChangeHeader(0, Long2Str(WindowNumber));
end;
{$F-}
begin
  {Open the stream file and register the data types}
  if not S.Init('STACKWIN.STM', SOpen , 1024) then begin
    WriteLn('Failed to Init stream, Status = ', InitStatus);
    Halt;
  end;
  S.RegisterHier(StackWindowStream);
  S.RegisterHier(ExplodingWindowStream);
  S.RegisterPointer(2000, @StackProc);

  {Allocate and read the windows from the stream}
  for I := 1 to 2 do begin
    Win[I] := WindowPtr(S.GetPtr);
    if Win[I] = nil then begin
      WriteLn('Stream error ', S.GetStatus);
      Halt;
    end;
  end;
  S.Done;

  {Draw the windows}
  for I := 1 to 2 do
    Win[I]^.Draw;
end.
```

Load

Instantiates two StackWindows from the stream created by the Store example. Since we know the stream contains exploding windows, we register the exploding window hierarchy. We also register the stack proc, StackProc, which changes the window's first header to display its window number.

The stream's GetPtr method calls StackWindow.Load indirectly to allocate heap space for the window and read its data. Note that a WindowPtr typecast is required to assign the result to the Win[I] variable.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

RawWindow.Load Store

Declaration
```
function OnStack : Boolean;
```

Purpose
Return True if window is displayed on the window stack.

Description
If the window is active, OnStack scans the window's stack from top to bottom. If the window is found there, OnStack returns True.

In an environment where the RawWindow and StackWindow types are mixed, OnStack may be valuable in distinguishing between the two. If all windows in the system are StackWindows and there is a single WindowStack, OnStack will return True if and only if IsActive is True.

Example
```
with Win[3]^ do
  if OnStack then
    Select
  else
    Draw;
```

If Win[3] is already on the stack, this will Select it. If it is not yet on the stack, this will draw it, add it to the stack and make it the current window. This is a contrived example since the Select method will automatically Draw a StackWindow if it isn't already on the window stack.

See Also
RawWindow.IsActive Select

Declaration
```
procedure Select; virtual;
```

Purpose
Make a window the current one by pulling to the top of stack.

Description
If the window is already current, Select just updates its contents by calling UpdateContents. If the window is not active, Select calls StackWindow.Draw. Otherwise, Select activates a virtual screen, erases the windows covering the desired one, erases the desired one, redraws the other windows, redraws the selected one, and deactivates the virtual screen.

Select also adds special logic that handles the case where a RawWindow (not on the stack) is current at the time of the call to Select. After activating the virtual screen, Select first redraws the top stacked window to make it the current one. This has the effect of deactivating the RawWindow (switching to its alternate frame). Then the normal Select logic, described in the previous paragraph, can take over.

Example
```
I := 1;
while ReadKeyWord <> $011B do begin
  Win[I]^.Select;
  Inc(I);
  if I > 3 then
    I := 1;
end;
```

Selects one of three windows every time a key is pressed.

See Also
Draw

SetStackProc

Declaration

```
procedure SetStackProc(SP : StackWindowProc);
```

Purpose

Set window stack proc.

Description

The stack proc is called by the Draw and DrawHidden methods just prior to actually stacking and drawing a given window. The main justification for using a custom stack proc is to display the window's number as part of a header. The window number is assigned just prior to calling the stack proc. The DESKPOP demonstration program uses this technique.

The procedure passed as a parameter to SetStackProc must be compiled with the FAR model, must be a global routine, and must have parameters that match those given by the StackWindowProc type.

Example

```
uses
  OpString, OpCrt, OpRoot, OpFrame, OpWindow;
var
  Win : array[1..3] of WindowPtr;
  WinOpts : LongInt;
  I : Integer;

  {$F+}
  procedure UpdateWindowNumber(WPtr : WindowPtr);
  begin
    with WPtr^ do
      ChangeHeader(0, Long2Str(WindowNumber));
  end;
  {$F-}

begin
  WinOpts := wBordered+wClear+wSaveContents;
  New(Win[1], InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
  New(Win[2], InitCustom(35, 8, 55, 18, DefaultColorSet, WinOpts));
  New(Win[3], InitCustom(40, 13, 60, 23, DefaultColorSet, WinOpts));
  for I := 1 to 3 do
    with Win[I]^ do begin
      wFrame.AddHeader(' ', heTC);
      SetStackProc(UpdateWindowNumber);
    end;

  {Switch between the windows}
  I := 1;
  repeat
    Win[I]^.Select;
    Inc(I);
    if I = 4 then
      I := 1;
  until ReadKeyWord = $011B;  {Escape}
end.
```

Supplies a StackWindowProc named UpdateWindowNumber. This procedure will update header number 0 using the window number it obtains from the WindowNumber function.

See Also

AssignNumber
Draw
WindowNumber

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store to stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the StackWindow to the stream S. If you have set a non-default stack proc for the window, you should be sure to register its address prior to calling this method (as shown in the example below). Store first calls RawWindow.Store to write the bulk of the window information, then writes the window's current number and the pointer code for the stack proc.

To check for errors that may occur during the Store, call the stream's GetStatus function.

The stream registration procedure for a StackWindow is StackWindowStream.

Example

```
uses OpString, OpCrt, OpRoot, OpFrame, OpWindow;
var Win : array[1..2] of WindowPtr;
    S : BufIdStream;
    I : Word;
    WinOpts : LongInt;

{$F+}
procedure StackProc(WPtr : WindowPtr);
begin
  with WPtr^ do
    ChangeHeader(0, Long2Str(WindowNumber));
end;
{$F-}

begin
  WinOpts := wBordered+wClear+wSaveContents;
  New(Win[1], InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
  New(Win[2], InitCustom(35, 8, 55, 18, DefaultColorSet, WinOpts));
  if (Win[1] = nil) or (Win[2] = nil) then begin
    WriteLn('Failed to Init StackWindow');
    Halt;
  end;
  for I := 1 to 2 do
    with Win[I]^ do begin
      wFrame.AddHeader(' ', heTL);
      EnableExplosions(20);
      SetStackProc(StackProc);
      Draw;
    end;

  {Write the windows to a stream}
  S.Init('STACKWIN.STM', SCreate , 1024);
  S.RegisterHier(StackWindowStream);
  S.RegisterHier(ExplodingWindowStream);
  S.RegisterPointer(2000, @StackProc);
  for I := 1 to 2 do
    S.PutPtr(Win[I]);
  S.Done;
end.
```

Store

Instantiates two StackWindow objects, adds some features, and then writes them to a stream. Because the windows are designed to explode, we register the exploding window stream as well as the required stack window stream. Because the windows use a non-default stack proc, the address of this routine is also registered.

See BufIdStream (in OpRoot) for proper error checking using streams.

See Also

Load

RawWindow.Store

Declaration

```
constructor wCopy(var S : StackWindow);
```

Purpose

Initialize a window from another one, making a unique copy of all dynamic structures.

Description

Works just like RawWindow.wCopy except that it copies the StackWindow's added data fields, including the active stack pointer and stack proc. The window number of the new window is always set to zero, since it is not initially active after the call to wCopy.

Example

```
WinOpts := wBordered+wClear+wSaveContents+wResizeable;
New(Win[1], InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
with Win[1]^.wFrame do begin
  AddHeader('Header', heTC);
  AddShadow(shBR, shOverWrite);
  AddScrollBar(frRR, 1, 9, DefaultColorSet);
end;
New(Win[2], wCopy(Win[1]^));
if Win[2] = nil then begin
  WriteLn('Failed to wCopy window, Status = ', InitStatus);
  Halt;
end;
```

Allocates and initializes Win[2] as an exact duplicate of Win[1], including all window and frame features.

See Also

RawWindow.wCopy

Declaration

```
function WindowNumber : Word;
```

Purpose

Return window number. 0 if inactive, else 1 or greater.

Description

When each StackWindow is drawn, it is assigned a unique integer number. If this number is displayed in a header of each window, it may form an integral part of the user interface design, helping the user to identify and switch between the windows on screen.

WindowNumber returns 0 for all inactive windows. Active windows receive sequentially larger window numbers, assigned in the order in which they are displayed. When a window is erased, its window number is returned to a free pool. The next window to be drawn gets the smallest window number in the free pool. The largest possible window number is 2047.

Example

```
Wnum := Win[1]^.WindowNumber;
```

Returns the window number that was assigned to Win[1] when it was originally stacked.

See Also

AssignNumber
Draw
SetStackProc

3. WindowStack

The WindowStack works in concert with the StackWindow to manage window relationships. There are only a few WindowStack methods you may want to call directly. Of course, because the WindowStack is derived from the more general PointerStack object (described in the OPROOT section), you can call any of those methods as well.

One method in WindowStack is of particular interest to applications that manage multiple windows simultaneously and that include mouse support. EvaluateStackMousePos determines which of the active windows, if any, the mouse falls into and returns a pointer to it. The GetPosResults method of the window can then return more detailed information: whether the mouse cursor is on the frame, within a scroll bar, and so on.

OPWINDOW declares a single global WindowStack variable (actually a typed constant) that it uses for all operations. The variable is named wStack and you'll typically refer to it when calling WindowStack methods. Because wStack is a typed constant, it's initialized automatically and there is no need to call its constructor.

Because the StackWindow object currently offers no legitimate way to use any stack other than wStack, you are unlikely to declare your own variables of type WindowStack.

The following program shows some of the capabilities available by using the WindowStack.

```
program WindowStackExample;  {EXWSTACK.PAS}
uses OpCrt, OpFrame, OpWindow;
var
  Win : WindowPtr;
  WinOpts : LongInt;
  I, J : Integer;
  S : String;
begin
  {Make three framed, resizable StackWindows}
  WinOpts := wBordered+wClear+wSaveContents+wResizeable;
  New(Win, InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
  Win^.Draw;
  New(Win, InitCustom(35, 8, 55, 18, DefaultColorSet, WinOpts));
  Win^.Draw;
  New(Win, InitCustom(40, 13, 60, 23, DefaultColorSet, WinOpts));
  Win^.Draw;
  Delay(1000);

  {Demonstrate the use of default wStack}
  for I := 1 to wStack.SP do begin
    Win := WindowPtr(wStack.Peek(I));
    for J := 2 to 10 do
      Win^.wFastWrite('Hello Mars', J, 5, TextAttr);
    Delay(1000);
  end;
  repeat
    Win := wStack.UnStackTop;
    if Win <> nil then begin
      Dispose(Win, Done);
      Delay(1000);
    end;
  until Win = nil;

  wStack.Done;
end.
```

Here we allocate and initialize three StackWindow objects. Note that we don't keep any individual record of each window's pointer. Instead, the windows are drawn immediately after allocation, so that their pointers are stored on the WindowStack.

Next we use the SP and Peek methods that wStack inherits from PointerStack. SP returns the index of the topmost stack element, and Peek returns the pointer at the requested location on the stack. Element 1 is the first window pushed. Note the WindowPtr typecast required to convert the pointer returned by Peek to the appropriate type. We call the StackWindow's wFastWrite routine to write some text to each of the windows; this is possible even though the first two windows are partially obscured.

The WindowStack's UnstackTop method erases the topmost window on the stack and pops the stack. It returns a pointer to the window it erased, or nil if there were no windows on the stack. We use the pointer to deallocate each StackWindow object in turn.

Finally, the program disposes of the WindowStack itself. In the example, this is just a gesture, because the program is about to halt anyway. If you're testing to see whether you've disposed of all heap space, it's important to do this. It's an easy step to forget since there is no call to wStack.Init in the program.

Constants

```
wStack : WindowStack = (
  psTop:0; psSize:0; psStatus:0; psBase:nil;
  wsScreen: (vsWidth:0; vsHeight:0; vsVideoSeg:0; vsSize:0;
             vsBufPtr:nil; vsAllocated:False);
  wsLevel:0 );
```

The default WindowStack, which all StackWindow objects use. The compiler initializes this object so that it's ready to use. The first time a window is pushed on the stack, the PointerStack object allocates a heap block large enough to hold InitStackSize windows. (InitStackSize is a constant, equal to 8, defined in OPROOT.) wStack dynamically expands itself if necessary.

Types

```
WindowStackPtr = ^WindowStack;
WindowStack =
  object(PointerStack)
    ...
  end;
```

The window stack object definition. Each WindowStack includes a data field of type VirtScreen. This virtual screen is used to manage the various operations on hidden windows as described in the previous section. The heap space for the virtual screen's buffer is not allocated until it is actually required and it is disposed as soon as it is not needed.

Done / EvaluateStackMousePos

Declaration
```
destructor Done; virtual;
```

Purpose
Clean up and dispose.

Description
Calls the Done methods of both the embedded virtual screen and PointerStack from which WindowStack is derived. The effect is to release all the heap space allocated by the WindowStack.

Example
```
wStack.Done;
```
Disposes of heap space used by the standard window stack, wStack.

See Also
PointerStack.Done VirtScreen.Done

Declaration
```
{$IFDEF UseMouse}
function EvaluateStackMousePos : WindowPtr;
```

Purpose
Evaluate mouse position using EvaluatePos.

Description
If a mouse is not installed, EvaluateStackMousePos always returns nil. Otherwise it determines the absolute position of the mouse cursor, passes this information to EvaluateStackPos, and returns the result from that routine.

This method is particularly useful in mouse-driven multi-window applications to allow the mouse to transfer control from one window to another. The DESKPOP demonstration program gives an example of its capabilities.

Example
```
Win := wStack.EvaluateStackMousePos;
if Win <> nil then begin
  UserCode := Win^.PosResults(FramePos, HotCode);
  {...Process results...}
end;
```
If Win is not nil, it is a pointer to an active StackWindow within which the mouse cursor is currently located. Calling that window's PosResults method returns more detail about the location of the mouse cursor relative to the window. See RawWindow.PosResults for more information.

See Also
EvaluateStackPos RawWindow.PosResults

Declaration

```
function EvaluateStackPos(X, Y : Byte) : WindowPtr;
```

Purpose

Evaluate position and return pointer to indicated window, if any.

Description

EvaluateStackPos scans from the topmost active StackWindow to the lowest one, calling each window's EvaluatePos method to determine whether the specified absolute (X, Y) position falls within the window. It returns a pointer to the first such window it finds. (More specifically, EvaluateStackPos looks for the first window where PosResults does not return frOutsideFrame in its first parameter.) If no window is found, EvaluateStackPos returns nil.

Note that EvaluatePos has already been called for the returned window, so that its PosResults method can be called immediately thereafter.

Example

```
X := 40;
Y := 10;
Win := wStack.EvaluateStackPos(X, Y);
if Win <> nil then begin
  UserCode := Win^.PosResults(FramePos, HotCode);
  {...Process results...}
end;
```

Returns a pointer to the highest window on wStack that includes the screen location (40, 10). If a valid window pointer is returned, the location is evaluated in more detail. See RawWindow for complete examples of EvaluatePos and PosResults.

See Also

EvaluateStackMousePos
RawWindow.EvaluatePos
RawWindow.PosResults

Declaration

```
constructor Init(Max : Word);
```

Purpose

Allocate stack and initialize.

Description

Allocates space to manage Max active windows. Like PointerStack, the WindowStack will automatically increase its size on demand. The downside of this automatic sizing is that it fragments the heap, so it's a good idea to choose Max large enough to hold all the windows.

The default WindowStack, wStack, has an initial size of InitStackSize, which is a constant, equal to 8, defined in OPROOT.

Init also initializes the WindowStack's virtual screen, but allocates no heap space for it at this time.

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load from stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to WindowStack.Store.

Load uses PointerStack.Load to do most of its work. This method allocates as much space for the stack as it had when Store was called, then polymorphically Loads each of the objects that were pushed on the stored stack and pushes pointers to each of them onto the new stack.

Because the WindowStack stores pointers to StackWindows, special types associated with StackWindow, such as those for exploding windows or stack procs, must be registered in order to successfully Load a WindowStack stream. The stream registration procedure for a WindowStack, WindowStackStream, does register the StackWindow type itself.

After the WindowStack is allocated, the objects are loaded, and the objects have been pushed onto the stack, WindowStack redraws each of the StackWindows in the order indicated by the stack.

If Load fails for any reason, InitStatus will hold a non-zero status code.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  S : BufIdStream;
begin
  {Load wStack from a stream}
  S.Init('WINSTACK.STM', SOpen, 1024);
  S.RegisterHier(WindowStackStream);
  S.Get(wStack);
  if S.GetStatus <> 0 then
    Halt;
  S.Done;

  {All the windows have been initialized and drawn in the original order}
end.
```

Loads the global wStack from the stream created in the Store example. Both of the StackWindows in the stream are allocated, initialized, and drawn. The stream's Get method calls WindowStack.Load indirectly.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

PointerStack.Load
StackWindow.Load
Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store to stream. Stores all currently stacked windows.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the WindowStack to the stream S. Prior to calling this method, you should be certain that the StackWindow type itself and also all types associated with StackWindows are registered with the stream manager.

Note that Store writes not only the WindowStack itself, but also all the windows currently pushed onto the stack. This behavior is useful when you want to store the entire state of a multi-window application, perhaps to be restored the next time the program is invoked.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

The stream registration procedure for a WindowStack is WindowStackStream.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  Win : WindowPtr;
  S : BufIdStream;
  WinOpts : LongInt;
  I, J : Word;
begin
  {Make two bordered StackWindows, push onto wStack}
  WinOpts := wBordered+wClear+wSaveContents+wStoreContents;
  New(Win, InitCustom(30, 3, 50, 13, DefaultColorSet, WinOpts));
  Win^.Draw;
  New(Win, InitCustom(35, 8, 55, 18, DefaultColorSet, WinOpts));
  Win^.Draw;

  {Fill the windows with text}
  for I := 1 to 2 do begin
    Win := WindowPtr(wStack.Peek(I));
    for J := 2 to 10 do
      Win^.wFastWrite('Hello Mars', J, 5, TextAttr);
    Win^.SaveContents;
  end;

  {Write the window stack and the windows to a stream}
  S.Init('WINSTACK.STM', SCreate, 1024);
  S.RegisterHier(WindowStackStream);
  S.Put(wStack);
  S.Done;

  Win := wStack.UnStackTop;
  while Win <> nil do begin
    Dispose(Win, Done);
    Win := wStack.UnStackTop;
  end;
end.
```

Store

Writes the global wStack to a stream. The stream's Put method calls WindowStack.Store indirectly. Both of the windows currently on wStack will also be stored (including their contents). See the Load example for a program to load this stream.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Load
PointerStack.Store
StackWindow.Store

Declaration
```
function TopWindow : WindowPtr;
```

Purpose
Return a pointer to the window at top of stack.

Description
TopWindow is simply shorthand for writing "WindowPtr(PeekTop)." It returns nil if the stack is empty.

Example
```
Win := wStack.TopWindow;
```
Returns a pointer to the window at the top of wStack (the default WindowStack) without changing the stack. Returns nil if wStack is empty.

See Also
PointerStack.Peek PointerStack.PeekTop

Declaration
```
function UnstackTop : WindowPtr;
```

Purpose
Erase and unstack window at top of stack.

Description
If the window stack is empty, this function returns nil and does nothing else. Otherwise it erases the window at the top of the stack, pops the stack, and returns a pointer to the window just erased. If an error occurs while erasing, UnstackTop also returns nil.

Example
```
Win := wStack.UnStackTop;
```
Erases the top window on wStack (the default WindowStack), pops it from the stack and returns a pointer to that window.

See Also
PointerStack.Pop StackWindow.Erase

4. CommandWindow

The CommandWindow is the first object in the window hierarchy to integrate keyboard and mouse input with the display of text. Useful windows often require some form of interaction with the user, even if it's only the requirement to "Press <Esc> to continue." To support this capability, CommandWindow incorporates a pointer to a CommandProcessor object (described in Chapter 3), which maps keystrokes to logical commands.

CommandWindow's most significant method is named Process. When you call Process, the window essentially takes control of the keyboard to process logical commands (scrolling around a pick list, for example) until the user enters one of the window's exit commands. At that time the Process method exits and the calling program can inspect the last keystroke and command entered in order to decide what to do next.

In one important respect, CommandWindow is an abstract object. Since it is impossible for it to implement a "universal" Process method, the one it does provide must be overridden by objects derived from it. Writing the Process method for a full-featured object like a text editor or pick list is a big task; fortunately for you, we've done that for the most common user interface capabilities. For a simpler case like a popup status window the Process method is correspondingly easier; we'll give an example of one shortly.

CommandWindow provides two additional hooks that fit nicely with its main purpose. The first is error handling capability. It stores a pointer to a procedure that is called automatically whenever an error occurs. By taking advantage of this pointer your application can centralize its error handling for all windows.

The second is a link to the OPHELP context-sensitive help system. Each CommandWindow stores an OPHELP topic index number of your choice. Whenever the user activates a help command, the Process method calls a standard help routine (specified by the window's CommandProcessor) using that topic number.

We'll show how to use these hooks in a moment. First, let's give an example of how to create your own CommandWindow.

```
program CommandWindowExample;  {EXCMDWIN.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow;
const
  {Define a trivial KeySet of a few cursor commands}
  KeyMax  = 18;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys          command type    key sequence}
  3,      $00, $48,     ccUp,           {Up}
  3,      $00, $50,     ccDown,         {Down}
  3,      $00, $4B,     ccLeft,         {Left}
  3,      $00, $4D,     ccRight,        {Right}
  2,      $1B,          ccQuit,         {Esc}
  0);                                   {Terminate with zero}
type
  SampleWindow =
    object(CommandWindow)
      procedure Process; virtual;
    end;
var
  Commands : CommandProcessor;
  CmdWin : SampleWindow;
```

```
  Finished : Boolean;

  procedure SampleWindow.Process;
  begin
    repeat
      {Get a command}
      GetNextCommand;
      case GetLastCommand of
        ccUp :    WriteLn('ccUp');
        ccDown :  WriteLn('ccDown');
        ccLeft :  WriteLn('ccLeft');
        ccRight : WriteLn('ccRight');
        ccQuit :  WriteLn('ccQuit');
        ccChar :  WriteLn('ccChar: ', Char(Lo(GetLastKey)));
        else      WriteLn('ccNone');
      end;
    until (GetLastCommand = ccQuit) or (GetLastCommand = ccError);
  end;

begin
  {Make a small CommandProcessor}
  Commands.Init(@KeySet, KeyMax);

  {Make a bordered CommandWindow}
  if not CmdWin.InitCustom(30, 5, 50, 15,                {Window coordinates}
                           DefaultColorSet,              {Color set}
                           wBordered+wClear+wSaveContents, {Window options}
                           Commands,                     {Command processor}
                           ucNone)                       {Unit code}
  then begin
    WriteLn('Failed to init CommandWindow. Status = ', InitStatus);
    Halt;
  end;

  {Add headers and draw window}
  CmdWin.wFrame.AddHeader(' Command Window ', heTC);
  CmdWin.wFrame.AddHeader(' <Esc> to Quit ', heBC);
  CmdWin.Draw;

  {Get and process commands}
  Finished := False;
  repeat
    CmdWin.Process;
    case CmdWin.GetLastCommand of
      ccQuit : Finished := True;                         {Quit}
      ccError : begin                                    {Error}
                  WriteLn('Error: ', CmdWin.GetLastError);
                  Finished := True;
                end;
      ccUser0..ccUser55 : WriteLn('user command');       {Handle exit command}
    end;
  until Finished;

  {Clean up}
  CmdWin.Done;
  Commands.Done;
end.
```

SampleWindow is a very simple extension of CommandWindow that overrides just the Process method. To prepare this type, we first define a KeySet typed constant as required for the command processor. Keys defined in this array will map to the specified logical commands;

other keys will map to ccNone. The example declares a CommandProcessor variable, Commands, to associate with the SampleWindow type, as well as the CmdWin variable itself.

SampleWindow's Process method gets commands by calling the CommandWindow's GetNextCommand method, which waits for keystrokes or mouse action (no mouse in this case, though). The Process example then checks which command was entered by calling GetLastCommand, and branches to different sections of code depending on the command. Here we just write out which command was entered. With two exceptions, the commands are the same as those in the KeySet. ccChar is a predefined command meaning a normal ASCII character (#32..#126, #128..#255). In this case, we can write out the character itself using the GetLastKey function. ccNone means that the keystroke matched no other defined command. Process loops until either the <Esc> key (ccQuit) or an error (ccError) is detected. ccError will never occur in the example because SampleWindow never sets it; nevertheless it's a good idea to get in the habit of checking for ccError.

The example first initializes the CommandProcessor to use the KeySet. Next it initializes the CommandWindow. Parameters to this constructor include the standard ones for any window–coordinates, colors, and options–plus two new ones–the CommandProcessor and a unit code. The unit code will be passed to the standard help and error handling procedures to help identify the source of the call. For this example it isn't important so we pass the predefined constant ucNone.

Like any other window, we can add headers using the methods from Frame and RawWindow.

After drawing the window, we enter a command handling loop which calls the Process method and checks the last command upon return. The last command will always be some form of exit command; in this case ccQuit is the only one that can actually occur. Other possibilities include ccError and the user-defined exit commands ccUser0..ccUser55. See CommandProcessor (Chapter 3) for further information.

The program ends by destroying the CommandProcessor and CommandWindow.

Error Handling Hook

Objects derived from CommandWindow call its GotError method whenever they detect an error. GotError takes two parameters, an error code and a message string, which is an optional convenience for the error handler. GotError in turn stores the error code in one of CommandWindow's data fields where it can be accessed by the GetLastError method. Then GotError calls a user-defined procedure, the error hook, passing it the error code, the error message, and a unit code. The default error procedure, DefaultErrorProc (declared in OPCMD), simply beeps and returns.

GotError then re-evaluates the error code, which the user-defined routine can change. If the error is not classified as "fatal," GotError resets the stored error code to zero. In **Object Professional**'s terminology, fatal means that the specific object instance that generated the error is no longer suitable for use. It doesn't necessarily mean that the program must halt. GotError's reclassification of error codes means that the error hook procedure is responsible for notifying the user of non-fatal errors and warnings, taking corrective action if possible, and setting the error type to fatal if correction is not possible.

It's easy to write and install a custom error procedure:

```
{$F+}
procedure MyErrorProc(UnitCode : Byte; var ErrorCode : Word;
                      ErrorMsg : string);
  {-Custom error handler}

var
  KW : Word;
begin
  if ErrorCode mod 10000 <> 0 then begin
    {Build an error message}
    if ErrorMsg = '' then
      ErrorMsg := 'Error '+Long2Str(ErrorCode);
    ErrorMsg := ErrorMsg+', Press a key to continue...';

    {Write it to the bottom of the screen and beep}
    FastWrite(ErrorMsg, ScreenHeight, 1, TextAttr);
    RingBell;

    {Wait for a key}
    KW := ReadKeyWord;

    {Treat all errors as fatal but let warnings go}
    if ErrorCode div 10000 = etNonFatal then
      ErrorCode := epFatal+(ErrorCode mod 10000);
  end;
end;
{$F-}
...
CmdWin.SetErrorProc(MyErrorProc);
```

A few words of review on **Object Professional**'s error codes are appropriate here. Codes are the sum of an error prefix, which is a multiple of 10000, and an error number, which ranges from zero to 9999. An error number of zero always means "no error." An error prefix of 0 is a fatal error, 10000 is a non-fatal error, 20000 is a warning, and 30000 is a non-error message.

This error procedure first assures that an error has actually occurred. If so, it builds an error message. Some objects pass an ErrorMsg string, so MyErrorProc uses that string if available. Otherwise it builds a string based on the error code. Then it writes the message, beeps the speaker, and waits for a keypress. This error handler also takes a conservative approach by reclassifying any non-fatal error into a fatal one so that the application will be able to read the error code using GetLastError.

The CommandWindow's SetErrorProc method is used to install the error handler. Each instance of a CommandWindow (or any derivative) can have its own unique error handler, or all of them can share a single one. The UnitCode parameter passed to the error handler can help to distinguish between calls from different objects.

A valid error procedure must be declared FAR using {$F+}, it must not be nested inside any other procedure, and it must take exactly the parameters shown in the example above.

Context Sensitive Help Hook

The CommandWindow's help hook builds upon the one provided by the CommandProcessor. That hook is described with OPCMD in Chapter 3. The new contribution is to provide a default help index for each CommandWindow. When a CommandWindow is initialized, its default help index is set to zero, which, at least for OPHELP, means no help topic at all. You can change the help topic to another value with the SetHelpIndex method, for example,

```
CmdWin.SetHelpIndex(100);
```

Some of **Object Professional**'s CommandWindow derivatives provide alternate techniques for getting the help index. The Menu object, for example, has a flag called mnUseItemForTopic. When this flag is set, the active menu item's code is passed to the help routine as the help index, overriding any value stored with SetHelpIndex. A similar option, pkUseItemForTopic, is available for the PickList object.

You can assign a CommandProcessor help procedure for a given window as follows:

```
CmdWin.GetCommandProcessor^.SetHelpProc(MyGetHelpProc);
```

Keep in mind that each CommandProcessor can have only one help hook at a time. See OPCMD for more information regarding the context sensitive help hook.

Types

```
ByteSet = set of Byte;
ByteSetPtr = ^ByteSet;
```

Each CommandWindow contains a field that points to a ByteSet. Valid members of this set are OPCMD command codes. Whenever the Process method of the CommandWindow detects the entry of one of these codes, it will exit just as if a normal exit command had been pressed. This capability is provided to aid in combining complex CommandWindow objects. For example, the Process method of a PickList object embedded in an EntryScreen should exit when the Tab key is pressed, consistent with other data fields in the EntryScreen. By adding the ccTab command code to the ByteSet, we can force the PickList to adopt this non-standard behavior temporarily. See the SetExitCommands method for more information.

```
CommandWindowPtr = ^CommandWindow;
CommandWindow =
  object(StackWindow)
    ...
  end;
```

Data types describing the CommandWindow object.

Declaration
```
function ClassifyError(ErrorCode : Word) : Byte;
```

Purpose
Return the class of the specified error code.

Description
If the value of (ErrorCode mod 10000) is zero, ClassifyError returns etNoError, indicating no error. If the ErrorCode is greater than or equal to 40000, ClassifyError returns etFatal, since there are no valid error codes that large. Otherwise, ClassifyError returns (ErrorCode div 10000). See Appendix D, Error Codes, for more information.

Example
```
Finished := False;
repeat
  CmdWin.Process;
  case CmdWin.GetLastCommand of
    ccQuit : Finished := True;                    {Quit}
    ccUser0..ccUser55 : WriteLn('user command'); {Handle exit command}
    ccError :                                     {Error}
      begin
        Status := CmdWin.GetLastError;
        if CmdWin.ClassifyError(Status) = etFatal then begin
          WriteLn('Error during Process, Status = ', Status);
          Finished := True;
        end;
      end;
  end;
until Finished;
```

Determines whether the error that caused Process to exit was fatal, and if so reports it and exits from the processing loop.

See Also
GetLastError

Declaration
```
function GetCommandProcessor : CommandProcessorPtr;
```

Purpose
Return pointer to window's command processor.

Description
Usually the CommandProcessor for a given CommandWindow is known in advance. (It must be passed as a parameter to the CommandWindow's constructor.) However, if the value is no longer available, the GetCommandProcessor function will return a pointer to the CommandWindow's CommandProcessor.

Example
```
CmdWin.GetCommandProcessor^.AddCommand(ccUser0, 1, $4400, 0);
```

Add the <F10> key ($4400) as exit command ccUser0 to the window's command processor.

See Also
SetCommandProcessor

GetExitCommandPtr / GetHelpIndex

Declaration
```
function GetExitCommandPtr : ByteSetPtr;
```

Purpose
Return pointer to set of special exit commands.

Description
If there are no special exit commands, GetExitCommandPtr returns nil. Otherwise it returns a pointer to a ByteSet that was originally specified with the SetExitCommands method.

This function will normally be called only in the Process method of a type derived from CommandWindow.

Example
```
if GetExitCommandPtr <> nil then
  if GetLastCommand in GetExitCommandPtr^ then
    Finished := True;
```

Determines whether the last command entered is in the set of special exit commands. If so, it sets a flag that will cause a loop to exit.

See Also
SetExitCommands

Declaration
```
function GetHelpIndex : Word;
```

Purpose
Return the help topic number for this window.

Description
For a freshly initialized CommandWindow, GetHelpIndex always returns zero. If another help index has been stored by calling the SetHelpIndex method, GetHelpIndex returns that value.

This function will normally be called only in the Process method of a type derived from CommandWindow.

Example
```
RequestHelp(GetHelpIndex);
```

Calls the CommandWindow's help hook routine using the help index stored for the window.

See Also
SetHelpIndex

Declaration

```
function GetLastCommand : Word;
```

Purpose

Return last command entered by user.

Description

For a freshly initialized CommandWindow, GetLastCommand returns ccNone. After a call to Process, GetLastCommand returns one of the window's normal exit commands (typically ccQuit, ccSelect, ccUser0..ccUser55, or ccError) or one of the commands in the special exit set (see SetExitCommands).

GetLastCommand may also be called within a Process method. In this case it can return any command code associated with the window's CommandProcessor. It is the Process method's responsibility to interpret all of these codes.

Calling GetLastCommand does not clear the stored value. It may be called as many times as needed.

Example

```
Finished := False;
repeat
  CmdWin.Process;
  case CmdWin.GetLastCommand of
    ccError,                                    {Fatal error}
    ccQuit : Finished := True;                  {Quit}
    ccUser0..ccUser55 : WriteLn('user exit');   {Handle exit command}
  end;
until Finished;
```

Uses GetLastCommand to determine what to do after Process.

See Also

GetLastKey SetLastCommand

Declaration

```
function GetLastError : Word; virtual;
```

Purpose

Return last error code and reset internal variable.

Description

If no error is pending, GetLastError returns zero. Otherwise it returns an error code which was stored when a method of the object called CommandWindow.GotError (and GotError didn't reset the error code) or when a lower level object such as RawWindow or even Frame detected a problem. GetLastError accesses all errors that can occur to a CommandWindow or derived type.

All error values can be interpreted as shown in Appendix D.

Although GetLastError resets its internal error value to zero, it is possible that other lower level errors may still be pending. To reset all pending errors, call ClearErrors.

Example

See the example program at the beginning of this section.

See Also

GotError RawWindow.ClearErrors

GetLastKey

Declaration

```
function GetLastKey : Word;
```

Purpose

Return last keystroke entered by user.

Description

In most cases the command code returned by GetLastCommand is enough for a Process method or application to decide what to do. After all, the main purpose of the CommandProcessor object is to map keystrokes to commands so that the keystrokes can be reinstalled without affecting the application.

There are at least two command codes that require further interpretation: ccChar and ccCtrlChar. These codes are returned when the user presses a normal alphanumeric key or control key, respectively. In the case of a TextEditor object, the ccChar command code means that the associated keystroke should be inserted into the associated text buffer. For a PickList object, ccChar usually means that the user wants to search for a pick item that starts with the character.

GetLastKey returns the full scan word of the keystroke. If the low byte (lo(GetLastKey)) is non-zero, it is the ASCII value of the key. The high byte of the word is the scan code, which is usually not of interest when GetLastKey is involved.

Calling GetLastKey does not clear the stored value. It may be called as many times as needed.

Example

```
repeat
  {Get a command}
  GetNextCommand;
  case GetLastCommand of
    ...
    ccChar :
      WriteLn('ccChar: ', Char(Lo(GetLastKey)));
  end;
until (GetLastCommand = ccQuit) or (GetLastCommand = ccError);
```

Within a Process method, detects an alphanumeric character and writes its ASCII value.

See Also

GetLastCommand

ReadKeyWord (OPCRT)

Declaration
```
procedure GetNextCommand; virtual;
```

Purpose
Called by Process to get next command from user.

Description
GetNextCommand calls the GetCommand function of the window's CommandProcessor. This function in turn calls the user-definable GetKeyProc to actually read from the keyboard or mouse. Hence, as described for the CommandProcessor, background tasks can occur while a CommandWindow is awaiting keyboard input.

GetNextCommand stores the returned command code and keystroke in internal variables, where they can be accessed by calls to GetLastCommand and GetLastKey.

Example
```
procedure SampleWindow.Process;
begin
  repeat
    GetNextCommand;
    case GetLastCommand of
      ...
    end;
  until (GetLastCommand = ccQuit) or (GetLastCommand = ccError);
end;

var CmdWin : SampleWindow;
...
CmdWin.GetCommandProcessor^.SetGetKeyProc(MyReadKey);
CmdWin.GetCommandProcessor^.SetKeyPressedProc(MyKeyPressed);
...
CmdWin.Process;
```

Sets CmdWin's user-definable keystroke routines to point to alternate routines as in the CommandProcessor example programs. CmdWin's GetNextCommand method will call these routines, perhaps allowing it to perform background tasks while awaiting input.

See Also
GetLastCommand GetLastKey

Declaration
```
procedure GotError(ErrorCode : Word; ErrorMsg : string);
```

Purpose
To be called when an error occurs.

Description
This method should be called only by methods of types derived from CommandWindow. It informs the object's error handler that an error has occurred and thus provides a user-defined routine an opportunity to handle the error. See "Error Handling Hook" above for more information.

ErrorCode must be consistent with the standards defined in Appendix D. It's not essential to pass an ErrorMsg string, but having one may be useful to the error hook routine.

Example
```
GotError(epFatal+1001, 'My application error message');
```

See Also
GetLastError

Init / InitCustom

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; var CP : CommandProcessor;
                 UnitCode : Byte);
```

Purpose

Create a command window with default options and colors.

Description

This constructor instantiates a CommandWindow using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
var
  Commands : CommandProcessor;
  CmdWin : SampleWindow;      {descendant of CommandWindow}
begin
  Commands.Init(@KeySet, KeyMax);
  if not CmdWin.Init(30, 5, 50, 15,      {Window coordinates}
                     Commands,           {Command processor}
                     ucNone)             {Unit code}
  then begin
    WriteLn('Failed to init CommandWindow. Status = ', InitStatus);
    Halt;
  end;
  ...
```

Creates a CommandWindow using default window options and colors. This window will use a CommandProcessor object named Commands.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet; Options : LongInt;
                       var CP : CommandProcessor; UnitCode : Byte);
```

Purpose

Create a command window with custom options.

Description

This constructor instantiates a CommandWindow using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window, where the window's text will be displayed. CP is an initialized CommandProcessor which will be used by the window's Process method to map keystrokes and mouse presses onto logical commands. UnitCode is a value that will identify the unit where the object resides to user-defined error and help handling routines. See OPROOT in Chapter 8 for a complete list of predefined unit codes. Codes in the range 100..255 are available for user-defined units.

InitCustom will fail only if the parent constructor (StackWindow.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Example

See the example program at the beginning of this section.

See Also

StackWindow.InitCustom

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a CommandWindow from stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to CommandWindow.Store. Load first calls StackWindow.Load to get much of the data; it then reads the data specific to a CommandWindow, including the window's help index, unit code, error procedure pointer, CommandProcessor pointer, and exit set pointer.

If the final three pointers were registered before Store was called, and they were registered again prior to calling Load, then Load will initialize the CommandWindow to use these same three pointers again. If no error procedure is registered, the CommandWindow will use the default, DefaultErrorProc. If no exit set pointer is registered, the CommandWindow will use an empty exit set.

If no CommandProcessor pointer is registered, things get a bit stickier. The CommandWindow absolutely must have a valid CommandProcessor before Process can be called. Even so, Load will not generate an error if it finds no CommandProcessor. In this way, the program can assign a CommandProcessor (by calling SetCommandProcessor) after the Load is complete. **Object Professional**'s own CommandWindow derivatives take this approach, which allows them to use a custom CommandProcessor if you've defined one or to use the standard CommandProcessor if not. Be warned, then, that no error will be detected until you call Process, ProcessAutoPilot, GetNextCommand, or RequestHelp if you forget to register the CommandProcessor pointer. When that occurs, you'll get the error epFatal+ecNoProcessor.

The stream registration procedure for a CommandWindow is CommandWindowStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow;
const
  {Define a trivial KeySet of a few cursor commands}
  KeyMax  = 18;
  KeySet  : array[0..KeyMax] of Byte = (
  {length keys          command type   key sequence}
  3,      $00, $48,     ccUp,          {Up}
  3,      $00, $50,     ccDown,        {Down}
  3,      $00, $4B,     ccLeft,        {Left}
  3,      $00, $4D,     ccRight,       {Right}
  2,      $1B,          ccQuit);       {Esc}
type
  SampleWindow =
    object(CommandWindow)
      procedure Process; virtual;
      constructor Load(var S : IdStream);
      procedure Store(var S : IdStream);
    end;
var
  Commands : CommandProcessor;
  S : BufIdStream;
  CmdWin : SampleWindow;
```

```
procedure SampleWindow.Process;
begin
  {Process commands}
end;

constructor SampleWindow.Load(var S : IdStream);
begin
  if not CommandWindow.Load(S) then
    Fail;
end;

procedure SampleWindow.Store(var S : IdStream);
begin
  CommandWindow.Store(S);
end;

{$F+}
procedure SampleWindowStream(SPtr : IdStreamPtr);
begin
  CommandWindowStream(SPtr);
  SPtr^.RegisterType(1000, 0, TypeOf(SampleWindow),
                     @SampleWindow.Store, @SampleWindow.Load);
end;
{$F-}

begin
  {Initialize CommandProcessor}
  Commands.Init(@KeySet, KeyMax);

  {Load a SampleWindow object from a stream}
  S.Init('CMDWIN.STM', SOpen, 1024);
  S.RegisterHier(SampleWindowStream);
  S.RegisterHier(ExplodingWindowStream);
  S.RegisterPointer(1000, @Commands);
  S.Get(CmdWin);
  S.Done;

  {Get and process commands}
  CmdWin.Draw;
  CmdWin.Process;
  CmdWin.Done;
  Commands.Done;
end.
```

Instantiates CmdWin from the stream created by the Store example. The CommandProcessor is instantiated normally and registered with a code of 1000.

An alternate way to assign the CommandProcessor to the SampleWindow would be to call SetCommandProcessor(Commands) within SampleWindow's Load method (after CommandWindow.Load returns). Using this approach you don't have to remember to register the pointer.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

CommandWindow.Load Store

Declaration

```
function PeekLastError : Word;
```

Purpose

Return last I/O error code without resetting internal variable.

Description

Works just like GetLastError, except that the internal status variable is not reset so that the same error code can be read again. This method is useful when writing a Process method for an object type derived from CommandWindow.

Example

See Process

See Also

GetLastError

Process

Declaration

```
procedure Process; virtual;
```

Purpose

Process commands.

Description

The Process method is the heart of the CommandWindow, but, ironically, if you call it you'll just generate a runtime error. That's because it's an abstract method which must be overridden by objects that make use of CommandWindow's capabilities.

We'll describe the general properties of the Process method; those that are shared by all valid implementations of it. Behavior specific to each implementation will be described along with the appropriate objects.

Applications call Process when they are ready to transfer control of the keyboard, mouse, and screen to the window that owns the method. Process is a general term that, depending on the window type, can mean Edit, Pick, GetMenuChoice, EditDataFields, ShowHelp, BrowseFile, etc. In fact, before the advantages of polymorphism became clear, **Object Professional**'s objects had methods by these names. By implementing them with one virtual method we were able to share more code, make their behavior more consistent, and open the door to extensible desktop managers like DESKPOP.

When a Process method gets control, it first checks for pending errors. If there are any, it exits immediately. Next it performs object-specific initialization, perhaps by checking the current window size and adjusting internal data structures as needed. After this initialization, Process draws the window if it is not already active, or updates its contents if it is. If the window is active but not current, Process generates an error (epNonFatal+ecWinNotCurrent) and exits. If this situation is possible, you should call Select before calling Process; for example:

```
if not MyWin.IsCurrent then
  MyWin.Select;
MyWin.Process;
```

Object Professional Process methods always turn OPCRT's break checking option (CheckBreak) off, so don't expect the <CtrlBreak> key to work inside of a Process method. If the mouse option of the associated CommandProcessor has been enabled, Process forces the mouse cursor to be visible.

Process then enters a command handling loop, where it calls GetNextCommand and performs object-specific processing for each command until an exit command is detected. Exit commands fall into four categories:

- standard exit commands ccQuit (exit and discard results if possible) and ccSelect (exit with a selection)
- user-defined exit commands ccUser0..$FFFF, which are generally treated like ccSelect. These commands can occur only if you've used CommandProcessor methods to assign keystrokes to them. See CommandProcessor.AddCommand for more information.
- dynamic exit commands assigned to the exit command set (see SetExitCommands).
- the special command ccError which indicates that a severe error occurred within Process or a method that it called.

Each **Object Professional** Process method provides consistent support for the ccHelp command, which indicates a user request for help. When this command is entered, Process calls the RequestHelp method, typically passing it the help index that was assigned to the CommandWindow with SetHelpIndex. Where it makes sense (for PickList and Menu objects specifically), there is also an option to pass the index of the currently selected item to RequestHelp instead.

When the command processing loop is finished, each Process method cleans up after itself, typically by storing its window state, restoring break checking, and hiding the mouse cursor if it was initially invisible. Note that Process does not erase the window, even if it was initially inactive.

A call to Process should be followed immediately by a case statement that branches based on the value of GetLastCommand. This will allow the application to take appropriate action based on how the user exited the window.

Example

```
procedure SampleWindow.Process;
var
  Finished : Boolean;
  SaveBreak : Boolean;
  MouseOn : Boolean;
begin
  {check for pending error}
  SetLastCommand(ccError);
  if PeekLastError <> 0 then
    Exit;
  {draw initial screen if not already done}
  Draw;
  if GetLastError <> 0 then begin
    GotError(epNonFatal+ecWinNotCurrent, emNullError);
    Exit;
  end;
  {turn break checking off}
  SaveBreak := CheckBreak;
  CheckBreak := False;
  {$IFDEF UseMouse}
  {force mouse cursor on if enabled, off if not}
  HideMousePrim(MouseOn);
  if GetCommandProcessor^.MouseEnabled then
    ShowMouse;
  {$ENDIF}
  {loop processing commands}
  Finished := False;
  repeat
    {get a command}
    GetNextCommand;
    {act on it}
    case GetLastCommand of
      ccError, ccQuit, ccSelect : Finished := True;
      ccUp :    WriteLn('ccUp');
      ccDown :  WriteLn('ccDown');
      ccChar :  WriteLn('ccChar: ', Char(Lo(GetLastKey)));
      ccHelp :  RequestHelp(GetHelpIndex);
      ccUser0..$FFFF : Finished := True;
    else
      {process dynamic exit commands}
      if GetExitCommandPtr <> nil then
        if GetLastCommand in GetExitCommandPtr^ then
          Finished := True;
    end;
  until Finished;
  CheckBreak := SaveBreak;
  ShowMousePrim(MouseOn);
end;
```

Implements a trivial process method that meets the **Object Professional** standards set forth above.

See Also

ProcessAutoPilot

ProcessAutoPilot

Declaration

```
procedure ProcessAutoPilot(var CmdList; NumCmds : Word);
```

Purpose

Process commands, starting with those in CmdList.

Description

CmdList is typically an array of Byte values, each element of which specifies a command that Process should evaluate. (The array would contain Word values if word-sized commands were activated in the window's CommandProcessor.) NumCmds specifies how many commands are in the array.

ProcessAutoPilot calls the CommandProcessor's SetCommandList method to activate these commands, then calls the object's Process method. Process takes commands from the specified array until that array is exhausted or an exit command is encountered. When the array is exhausted, Process reverts to using the keyboard or mouse.

See the section "Auto-Execution of Commands" in Chapter 3 (OPCMD) for more information.

If the CommandWindow was loaded from a stream and no CommandProcessor was registered, ProcessAutoPilot will generate the error epFatal+ecNoProcessor.

Because the CmdList parameter is untyped, the compiler cannot catch errors made in specifying it. The Process method will blindly attempt to execute the commands that correspond to whatever byte values are passed to it. Be careful!

Example

```
const
  CmdList : array[1..3] of Byte = (ccDown, ccDown, ccRight);
...
begin
  {...Init/draw a CmdWin...}

  Finished := False;
  repeat
    CmdWin.ProcessAutoPilot(CmdList, 3);
    case CmdWin.GetLastCommand of
      ccQuit, ccError : Finished := True;
      ccUser0..ccUser55 : WriteLn('user command');   {Handle exit command}
    end;
  until Finished;

  Commands.Done;
  CmdWin.Done;
end.
```

Passes a list of three commands to ProcessAutoPilot. Once CmdList is exhausted, ProcessAutoPilot works just like Process.

See Also

CommandProcessor.SetCommandList Process

Declaration

```
procedure RequestHelp(HelpIndex : Word); virtual;
```

Purpose

Called by Process to request context-sensitive help.

Description

RequestHelp calls the user help routine assigned to the CommandWindow's CommandProcessor. Typically this method is called only from within the Process method of the window.

RequestHelp passes the window's unit code, the instance's Self address, and HelpIndex to the user help routine. See the section entitled "Context Sensitive Help Hook" in the CommandProcessor section of Chapter 3 for more information.

If the window has no CommandProcessor associated with it (only possible when the CommandWindow was loaded from a stream), RequestHelp generates the error epFatal + ecNoProcessor.

Example

```
{get a command}
GetNextCommand;
{act on it}
case GetLastCommand of
  ccError, ccQuit, ccSelect :
    Finished := True;
  {...}
  ccHelp :
    RequestHelp(GetHelpIndex);
end;
```

Calls the help hook with the window's assigned help index.

Declaration

```
procedure SetCommandProcessor(var CP : CommandProcessor);
```

Purpose

Set command processor to use.

Description

A CommandProcessor is assigned to a CommandWindow when it is initialized. This method changes the CommandProcessor at a later time. SetCommandProcessor is essential when a CommandWindow has been loaded from a stream and no CommandProcessor was registered.

CP is an initialized CommandProcessor object. SetCommandProcessor stores just the address of the CommandProcessor within the CommandWindow. Hence, the specified CommandProcessor must remain valid during the lifetime of the CommandWindow. (For example, it cannot be a local variable that goes out of scope while the window still exists.)

Example

```
Commands.Init(@KeySet, KeyMax);
OtherCommands.Init(@OtherKeySet, OtherKeyMax);
CmdWin.Init(30, 5, 50, 15, Commands, ucNone);
{...}
CmdWin.SetCommandProcessor(OtherCommands);
```

Replaces the original CommandProcessor, Commands, with OtherCommands.

See Also

GetCommandProcessor Load
Init

SetErrorProc

Declaration

```
procedure SetErrorProc(EP : ErrorProc);
```

Purpose

Set error handler.

Description

When a CommandWindow is initialized, it is assigned a default error handler (DefaultErrorProc) which simply beeps when an error occurs. Use SetErrorProc to assign another error procedure to the CommandWindow.

The procedure EP must be declared FAR with the {$F+} directive and must not be nested inside of any other procedure or function. Its parameters must match those of the ErrorProc type, which is declared in the OPROOT unit.

See "Error Handling Hook" at the beginning of this section for more information.

Example

```
var
  Commands : CommandProcessor;
  CmdWin : SampleWindow;            {Derived from CommandWindow}

{$F+}
procedure MyErrorProc(UnitCode : Byte;
                      var ErrorCode : Word; ErrorMsg : string);
var
  KW : Word;
begin
  if ErrorCode <> 0 then begin
    RingBell;
    CmdWin.wFastFill(80, ' ', 1, 1, TextAttr);
    CmdWin.wFastWrite(ErrorMsg, 1, 1, TextAttr);
    KW := ReadKeyWord;
    CmdWin.wFastFill(80, ' ', 1, 1, TextAttr);
  end;
end;
{$F-}

begin
  {...Create/draw CmdWin...}
  CmdWin.SetErrorProc(MyErrorProc);
  CmdWin.Process;
  CmdWin.GotError(1, 'Test Error Message');
  {...}
end.
```

Hooks MyErrorProc into CmdWin. MyErrorProc will display a message whenever an error occurs. This example simulates an error (using GotError) when it returns from Process.

See Also

GotError

Declaration

```
procedure SetExitCommands(ExitSetPtr : ByteSetPtr);
```

Purpose

Specify set of special exit commands.

Description

There are two ways to assign special exit commands to a CommandWindow. The first is by using the window's CommandProcessor to assign keystroke sequences to the commands ccQuit, ccSelect, or ccUser0..ccUser55. ccUser0 and up are especially reserved for user exit commands and these will serve most needs.

The second technique is activated with the SetExitCommands method. Instead of working with keystroke sequences, SetExitCommands works with the logical commands themselves. It has an effect only with commands that are not already interpreted by the Process method of the CommandWindow. Command elements in the set of Byte pointed to by ExitSetPtr cause the Process method to exit as if a normal exit command had been entered. (At least that's how it works in **Object Professional**'s own Process methods. If you write your own CommandWindow derivatives, you could interpret the ExitSetPtr differently, perhaps giving it priority over the other commands.)

Let's make this more concrete. Suppose you have a Process method that interprets the following commands: ccUp, ccDown, ccLeft, ccRight, ccQuit, ccSelect, and ccUser0..$FFFF. The Process method will act upon these commands no matter what ExitSetPtr points to. Now suppose ExitSetPtr points to the set [ccTab, ccBackTab]. **Object Professional**'s Process methods will now exit whenever either of these commands is detected.

When is this useful? We added it primarily to enhance the flexibility of the PickList object, which is often embedded in other objects. In an EntryScreen, a PickList can act as a single data field. When this is so, we want ccTab and ccBackTab to force an exit from the PickList, just as the same commands cause the cursor to move between data fields of other types. It's not practical for us to hard-code ccTab and ccBackTab into the PickList's Process method without further complications. Instead we use SetExitCommands.

CommandWindow interprets an ExitSetPtr of nil the same as it would a pointer to an empty set. If ExitSetPtr is not nil, it must point to a set that remains valid throughout the life of the CommandWindow.

Even if word-sized commands are activated, ExitSetPtr will work only with the commands in the range 0..255, due to the limitations of Turbo Pascal sets. Besides, most **Object Professional** Process methods interpret the entire range ccUser0..$FFFF as used-defined exit commands.

Example

```
const
  MyExitCommands : ByteSet = [ccUpLeft..ccDownRight];
...
  CmdWin.SetExitCommands(@MyExitCommands);
```

Informs CmdWin of a list of commands that the Process method should treat as exit commands.

See Also

GetExitCommandPtr

SetHelpIndex / SetLastCommand

Declaration
```
procedure SetHelpIndex(Index : Word);
```

Purpose
Set help topic number for this window.

Description
The specified Index number is usually the same as a topic number in an OPHELP-compatible help system. This is not a requirement, however, since the index number is just passed along to a user defined help routine when the RequestHelp method is called.

See "Context Sensitive Help Hook" at the beginning of this section for more information.

Example
```
CmdWin.SetHelpIndex(1000);
```

See Also
GetHelpIndex RequestHelp

Declaration
```
procedure SetLastCommand(Cmd : Word);
```

Purpose
Change last command.

Description
This method is typically used only in Process methods to set the stored value of the last command. This occurs most often to store a value of ccError when an error has occurred, or to change ccMouse (left mouse click) into ccSelect (select and exit) when the mouse cursor is positioned correctly at the time of the click.

Example
See the Process method's example.

See Also
GetLastCommand

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store to stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the CommandWindow to the stream S. Store first calls StackWindow.Store to write much of the data; it then writes the data specific to a CommandWindow, including the window's help index, unit code, error procedure pointer, CommandProcessor pointer, and exit set pointer.

If you are using a non-default error procedure or exit set, you should register its address prior to calling this method. You should always register the address of the CommandProcessor unless you have created Store and Load methods that assign the CommandProcessor explicitly for your window type. See Load for more information.

The stream registration procedure for a CommandWindow is CommandWindowStream.

Example

```
{...same as Load...}

begin
  Commands.Init(@KeySet, KeyMax);
  WinOpts := wBordered+wClear+wSaveContents+wResizeable;
  if not CmdWin.InitCustom(30, 5, 50, 15, DefaultColorSet,
                           WinOpts, Commands, 0) then begin
    WriteLn('Failed to Init CommandWindow,  Status = ', InitStatus);
    Halt;
  end;
  CmdWin.wFrame.AddHeader('Command window ', heTC);
  CmdWin.EnableExplosions(20);
  CmdWin.Draw;

  {Write CmdWin out to a stream}
  S.Init('CMDWIN.STM', SCreate, 1024);
  S.RegisterHier(SampleWindowStream);
  S.RegisterHier(ExplodingWindowStream);
  S.RegisterPointer(1000, @Commands);
  S.Put(CmdWin);
  S.Done;
  CmdWin.Done;
  Commands.Done;
end.
```

Instantiates CmdWin (derived from CommandWindow) and stores it to the stream. We've registered our CommandProcessor with a code 1000. A program that loads this stream (see Load for an example) should create its own CommandProcessor and register a pointer to it with a code of 1000.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Load StackWindow.Store

5. LoadableColorSet

The LoadableColorSet is a straightforward extension of the ColorSet, which is described in Chapter 2 (OPCRT). The new object adds exactly three methods: Init, Load, and Store. The purpose of these methods, of course, is to allow reading and writing ColorSets in streams. There is no point in choosing a LoadableColorSet over a plain ColorSet unless you want to use streams.

A LoadableColorSet includes a normal ColorSet as a data field. This field can be used to access the data and methods of the ColorSet itself.

The following program demonstrates how to store a ColorSet in a stream.

```
program ColorSetExample;  {EXCOLOR.PAS}
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  MyLcs : LoadableColorSet;
  S : BufIdStream;
  Status : Word;
begin
  {Start off with the default color set}
  MyLcs.Init(DefaultColorSet);

  {Customize the LoadableColorSet}
  with MyLcs.lcsColors do begin
    SetTextAttr($17, $07);
    SetFrameAttr($16, $07);
    SetHeaderAttr($16, $07);
  end;

  {Make a Stream}
  if not S.Init('MYLCS.STM', SCreate, 1024) then begin
    Writeln ('Failed to Create Stream,  Status = ', InitStatus);
    Halt;
  end;
  S.RegisterHier(LoadableColorSetStream);
  S.Put(MyLcs);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Failed to Store MyLcs, Status = ', Status);
    Halt;
  end;
  S.Done;

  {MYLCS.STM is now available for use by other programs}
end.
```

This example initializes a LoadableColorSet (MyLcs) from the default color set provided by OPCRT, then changes some values. It creates a new stream file, registers the LoadableColorSet data type, and writes MyLcs to the stream. Another program can now read this LoadableColorSet (see the Load example below). In this way, streams provide an alternative mechanism for program installation and customization.

Types

```
LoadableColorSet =
  object(Root)
    lcsColors : ColorSet;
    ...
  end;
```

The data type for the LoadableColorSet. There is no access method provided for the lcsColors data field; you should refer to it directly. For example, if you need to pass a ColorSet to a window's InitCustom method, you should use the following syntax:

```
MyWin.InitCustom(X1, Y1, X2, Y2, MyLcs.lcsColors, Options);
```

If you need to modify the color fields in the ColorSet, use the following:

```
MyLcs.lcsColors.SetTextAttr($17, $07);
```

Init / Load

Declaration
```
constructor Init(var Colors : ColorSet);
```

Purpose
Initialize this color set from another one.

Description
This constructor calls Root.Init and then assigns Colors to the lcsColors data field. It can fail only if Root.Init does.

Example
```
MyLcs.Init(DefaultColorSet);
```
Establishes DefaultColorSet as the initial color set for MyLcs.

See Also
Root.Init

Declaration
```
constructor Load(var S : IdStream);
```

Purpose
Load a color set from a stream.

Description
S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to LoadableColorSet.Store. Load simply calls Root.Init and then reads the lcsColors data field.

The stream registration procedure for a LoadableColorSet is LoadableColorSetStream.

Example
```
uses OpCrt, OpRoot, OpFrame, OpWindow;
var Win : WindowPtr;
    MyLcs : LoadableColorSet;
    S : BufIdStream;
begin
  {Load MyColorSet from a stream}
  S.Init('MYLCS.STM', SOpen, 1024);
  S.RegisterHier(LoadableColorSetStream);
  S.Get(MyLcs);
  S.Done;
  {Use the loaded color set, frame should be green on black}
  New(Win, InitCustom(30, 5, 50, 15, MyLcs.LcsColors,
                      DefWindowOptions or wBordered));
  Win^.Draw;
  Win^.wFastWrite('Hello World', 1, 1, TextAttr);
  Delay(1000);
  Dispose(Win, Done);
end.
```
Loads MyLcs from a stream created by the Store example. S.Get calls Load indirectly via the stream manager.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also
Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a color set in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store simply writes the lcsColors field directly to the stream.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

See BufIdStream (in OPROOT) for proper error checking using streams.

Example

See the introduction to this section.

See Also

Load

6. PackedWindow

The PackedWindow object serves a function somewhat similar to the ScreenRect object, except that it stores the contents of the designated screen rectangle in a "packed" format. More specifically, it stores the contents of the rectangle as a pair of arrays—one containing the characters, one containing the attributes—and performs run-length encoding on each of the two arrays. This technique allows any sequence of 6 or more identical bytes in the array to be stored as a 5-byte record. Space savings due to compression will typically range from 30 to 90%. Compression and decompression are performed by assembly language routines, leading to quite acceptable display speed.

PackedWindow includes a host of methods for initializing packed windows from video memory, displaying them in their original positions or new ones, saving them to files, and for reading them back.

PackedWindow is an object-oriented version of the same concept provided by Turbo Professional's TPCRT unit. In fact, PackedWindow provides special methods to read and write disk files compatible with Turbo Professional's packed windows. (The PSCREEN demonstration program, a TSR that captures any text screen to a file, uses the same packed file format. PSCREEN is described in the Utilities section of this chapter.)

PackedWindows can also be stored in streams, thus automatically providing the capability to manage random-access libraries of packed screens. Such libraries are ideal for applications that require large numbers of static display screens.

Here's an example that captures part of the screen in a PackedWindow and then restores it in a flash.

```
program PackedWindowExample;  {EXPACKW.PAS}
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  PWin : PackedWindow;
  I, J : Integer;
begin
  {Write some characters to the screen}
  for I := 1 to 25 do
    for J := 1 to 26 do
      FastWrite(Char(64+J), I, J+I, TextAttr);

  {Create a PackedWindow}
  if not PWin.Init(10, 5, ScreenWidth-10, ScreenHeight-5) then begin
    WriteLn('Failed to Init PackedWindow,  Status = ', InitStatus);
    Halt;
  end;

  {Clear the initial screen}
  ClrScr;
  FastWrite('Press any key to restore the PackedWindow', 13, 23, TextAttr);
  I := ReadKeyWord;

  {Display the PackedWindow, clean up}
  PWin.Display;
  PWin.Done;
end.
```

PWin.Init at first allocates a heap buffer large enough to hold an uncompressed version of the specified rectangle. It reads video memory at the appropriate locations, and compresses the

data into the buffer. Then it gives back the portion of the buffer that it didn't need. PWin.Display redraws the packed image at the original location. For this example the packed image takes about 25% the space that an unpacked buffer would. The run-length compression does an excellent job on the attribute portion of the window–because it's all drawn in the same attribute the whole thing reduces to 5 bytes. With the character portion it doesn't do as well because the alphabetic sequences cannot be compressed. The spaces at the corners of the window do compress nicely, lending to the overall compression.

Reading and Writing Packed Windows

PackedWindow includes two sets of routines for writing windows to disk and reading them back. Each set is a matched pair of routines which must be used consistently.

Packed windows written with the Write method must be read with the Read method. Read is compatible with packed window files created by Turbo Professional 5.0. Read also reads files created by the PSCREEN popup screen capture utility. The InitFromMemory constructor uses window images written by Write and bound into the program's EXE file using Borland's BINOBJ utility.

Windows written with Store must be read with Load. These methods are based on streams (see Chapter 8). Using streams, any number of objects (of the same or varying types) can be stored in a single file. Using streams you can also manage random-access libraries of PackedWindows.

Types

```
PackRec =
  record
    Link : Word;          {number of bytes to next PackRec}
    FillVal : Byte;       {fill value}
    FillCnt : Word;       {fill count}
  end;
```

Records like this specify runs of repeated bytes in the compressed screen. Because assembly language routines access the records, you should not change the PackRec definition without also changing the assembly code.

```
PackedScreen = array[1..4000] of Byte;
```

A mask for the packed array that PackedWindow writes and reads. The actual size varies depending on the window coordinates and compression ratio.

```
PackedWindowPtr = ^PackedWindow;
PackedWindow =
  object(Root)
    ...
  end;
```

The PackedWindow object. The organization of its internal data structures cannot be changed without invalidating both assembly language code and existing packed window files.

Display / DisplayAt

Declaration

```
procedure Display;
```

Purpose

Display the packed window.

Description

Draws a previously initialized packed window at its original location. Unlike the RawWindow.Draw method, Display does not save the underlying screen, set the OPCRT window coordinates, or position the cursor. Use the PackedWindow on top of a StackWindow if these features are needed.

Example

```
PWin.Init(1, 1, 70, 20);
...
PWin.Display;
```

Redraws the screen contents originally at (1, 1, 70, 20) in the same position.

See Also

MapColors

Declaration

```
procedure DisplayAt(Row, Col : Word);
```

Purpose

Display the packed window at Row, Col. If necessary, the coordinates are adjusted to allow it to fit on the screen.

Description

If Row or Col is zero, the value is increased to one. If either dimension of the packed window is larger than the current screen size, DisplayAt exits without doing anything (no error is generated). If the packed window would extend beyond the screen edge, the Row and Col coordinates are adjusted just enough to make the window fit on the screen. For example, if the window is 40 columns wide, and Col is 70, Col would be changed to 41 (on an 80 column display) to insure that the window fits.

Example

```
PWin.Init(1, 20, 80, 25);
...
PWin.DisplayAt(1, 1);
```

Save the last six rows of the screen as a packed window, then display them on the top six rows of the screen.

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate the packed screen.

Description
If the packed window was originally allocated on the heap (i.e., InitFromMemory was *not* used to initialize it), Done deallocates the heap buffer. Then it calls Root.Done.

Note that Done does not "erase" the packed window image or change the screen in any way.

Example
```
PWin.Init(1, 1, 70, 20);
...
PWin.Done;
```
Allocates a packed window and later deallocates it.

See Also
InitFromMemory

Declaration
```
constructor Init(XL, YL, XH, YH : Byte);
```

Purpose
Create a packed window.

Description
Init first allocates a heap buffer of 2*(XH-XL+1)*(YH-YL+1) bytes, which is the size required if no compression were used. If insufficient memory is available, Init stores the error epFatal+ecOutOfMemory in InitStatus and exits. Init will also exit if the specified coordinates are bad (XH is less than XL, for example).

Init then calls an assembly language routine which reads the current contents of video memory and packs first the characters then the attributes into the heap buffer.

Finally Init frees the unused portion of the heap buffer. Depending on how compressible the image was, the unused portion may be 30-90% of the original heap block.

Example
```
if not PWin.Init(1, 1, ScreenWidth, ScreenHeight) then begin
  WriteLn('Failed to Init PackedWindow,  Status = ', InitStatus);
  Halt;
end;
```
Instantiates PWin and packs the entire screen.

See Also
InitFromMemory Read
Load

InitFromMemory

Declaration

```
constructor InitFromMemory(P : Pointer);
```

Purpose

Initialize a packed window from one stored to disk with Write and bound into the EXE file with BINOBJ.

Description

This constructor provides a convenient way to use predesigned screens. The PSCREEN screen capture TSR (described in Section H.2) writes packed windows to disk in the correct format to use with InitFromMemory.

P specifies the address of the BINOBJ data block. InitFromMemory initializes its internal data from this data block and stores a pointer to the compressed image within the block. InitFromMemory does not allocate heap space for the screen image.

It is not absolutely necessary to call Done for PackWindow objects initialized with InitFromMemory. Even so, it's a good idea to do so.

Note that, unlike the Read constructor, InitFromMemory does not work with packed windows created by Turbo Professional 5.0. They must be converted to Object Professional format (using Read and Write) before running BINOBJ.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  PWin : PackedWindow;

  {$L PWMEM.OBJ}
  procedure PwMem; external;

begin
  {Initialize PWin from memory}
  if not PWin.InitFromMemory(@PwMem) then begin
    WriteLn('Failed to Init PackedWindow,  Status = ', InitStatus);
    Halt;
  end;

  {Display the PackedWindow, clean up}
  ClrScr;
  PWin.DisplayAt(1, 1);
  PWin.Done;
end.
```

This program has a packed window bound in at compile-time. The bound-in file was created by running Borland's BINOBJ program on files created with the Write method (in this case, using the example program for the Write method). The following command line was used:

```
BINOBJ PACKWIN.PW PWMEM.OBJ PWMEM
```

This creates a file named PWMEM.OBJ that has a public identifier PWMEM. PWMEM is referenced in the above example as the starting point for InitFromMemory. The $L directive instructs the compiler to link PWMEM.OBJ into the program. And the 'external' procedure means that the compiler will find the routine within the OBJ file. In this case it's not a procedure at all, but instead some data representing a compressed window. Don't try to call this "procedure!" Note that InitFromMemory refers to PwMem using the '@' operator, which returns the address of the fake procedure.

See Also

Write

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a packed window from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to PackedWindow.Store. Load reads the size of the packed window from the stream and allocates exactly enough heap space to read the packed image.

The stream registration procedure for a PackedWindow is PackedWindowStream.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  PWin : PackedWindow;
  S : BufIdStream;
begin
  {Load PWin from a stream}
  S.Init('PACKWIN.STM', SOpen, 1024);
  S.RegisterHier(PackedWindowStream);
  S.Get(PWin);
  S.Done;

  {Display the PackedWindow, clean up}
  PWin.Display;
  PWin.Done;
end.
```

Instantiates PWin from the stream created by the Store example using the stream's Get method, which calls Load indirectly.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Store

MapColors / Read

Declaration

```
procedure MapColors;
```

Purpose

Map the colors of the packed window for improved appearance on mono/B&W displays.

Description

Typically, this method is to be used in cases where a window was drawn on a color display, saved to a file, then read back from disk on another machine.

If the UseColor function (see OPCRT) returns True, MapColors exits immediately. Otherwise it remaps all of the window's attribute values using the ColorMap array in OPCRT. ColorMap is initialized to a reasonable mapping, but you still may want to change it; there's a lot of room for arbitrary decisions in mapping 128 colors to 10 (at most) attribute combinations on a monochrome display.

Note that calling this procedure permanently changes the attributes in the packed window, so if you save the window back to a file after remapping the colors, the original color information will be lost. A "reverse" mapping is not possible.

Example

```
PWin.Read('TEST.PW');
PWin.MapColors;
PWin.Display;
```

Read a packed window from disk, then map its colors for a monochrome video mode. Note that if the current video mode allows colors, MapColors will do nothing.

See Also

UseColor (OPCRT)

Declaration

```
constructor Read(FName : string);
```

Purpose

Read the packed window stored in FName into memory.

Description

This method reads into memory a packed window previously saved to disk by the Write method. (Note that PSCREEN writes packed windows this way.) I/O errors (file not found or a read error, most likely) will cause the constructor to fail; the error code is stored in InitStatus. Read also insures that the file contains a valid packed window. If not, InitStatus will contain epFatal+ecInvalidFormat.

Read gets the size of the packed window from the stream and allocates exactly enough heap space for the packed image.

Read uses logic that allows it to read packed windows stored by Turbo Professional 5.0 programs as well as those in the slightly different format used by **Object Professional**.

Example

```
if not PWin.Read('PACKWIN.PW') then begin
  WriteLn('Failed to Init PackedWindow,  Status = ', InitStatus);
  Halt;
end;
```

Reads the packed window file PACKWIN.PW.

See the Write example for a complete program that writes and reads a packed window.

See Also

InitFromMemory
Load
Write

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a packed window in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes data describing the packed window followed by the compressed image to the next location in the stream.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

See BufIdStream (in OPROOT) for proper error checking using streams.

The stream registration procedure for a PackedWindow is PackedWindowStream.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  PWin : PackedWindow;
  S : BufIdStream;
  I, J : Word;
begin
  {Write some characters to the screen}
  for I := 1 to 25 do
    for J := 1 to 26 do
      FastWrite(Char(64+J), I, J+I, TextAttr);

  {Create a PackedWindow of the entire screen}
  if not PWin.Init(1, 1, ScreenWidth, ScreenHeight) then begin
    WriteLn('Failed to Init PackedWindow,  Status = ', InitStatus);
    Halt;
  end;

  {Write the packed window to a stream}
  S.Init('PACKWIN.STM', SCreate, 1024);
  S.RegisterHier(PackedWindowStream);
  S.Put(PWin);
  S.Done;
  PWin.Done;
end.
```

Creates a packed window and stores it to a stream. See Load for an example program to load this stream.

See Also

Load

Write

Declaration

```
function Write(FName : string) : Word;
```

Purpose

Write the packed window to FName.

Description

Write returns a status code in the function result. For a successful write, the status is zero; otherwise an error code consistent with those described in Appendix D.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  PWin : PackedWindow;
  I, J : Integer;
  Status : Word;
begin
  {Write some characters to the screen}
  for I := 1 to 25 do
    for J := 1 to 26 do
      FastWrite(Char(64+J), I, J+I, TextAttr);

  {Create a PackedWindow of the entire screen}
  PWin.Init(1, 1, ScreenWidth, ScreenHeight);

  {Write out PWin and destroy it}
  Status := PWin.Write('PACKWIN.PW');
  if Status <> 0 then
    WriteLn('Failed to write PWin, Status = ', Status);
  PWin.Done;

  ClrScr;

  {Read PWin again}
  if not PWin.Read('PACKWIN.PW') then begin
    WriteLn('Failed to Init PackedWindow,  Status = ', InitStatus);
    Halt;
  end;
  PWin.Display;
  PWin.Done;
end.
```

Writes PWin to disk then reads it back again.

See Also

Read

7. VirtScreen

VirtScreen is short for virtual screen. This object provides a collection of methods for treating a region of memory as if it were the physical display screen. We can write to this region of memory, clear it, initialize it from the physical screen, and copy it back to the physical screen. These capabilities are useful when we want to build a complex display in the background and then make it visible in a split second.

An important feature of virtual screens is that they can be larger than the normal 25 line by 80 column display. VirtScreen objects can manage any screen whose total memory usage does not exceed 64K bytes. For example, a VirtScreen could have 80 columns and over 400 rows (accounting for the two bytes used by each character/attribute pair). By copying adjacent regions from a large virtual screen to the smaller physical one we create the illusion of very fast scrolling.

VirtScreen supplies its own methods to clear, write to, and read from a virtual screen. You cannot use Turbo Pascal's Write and WriteLn statements to write to it. (It is theoretically possible to make WriteLn work on a virtual screen. In fact, Turbo Professional provided a unit, TPSCREEN, which did exactly that. The disadvantages of the required technique outweighed the advantages, however, and TPSCREEN is now history.) VirtScreen provides an alternate technique for writing to virtual screens that is much more valuable because of its connection to other objects.

Whenever you call VirtScreen's Activate method, the video segment stored in OPCRT is temporarily pointed at the virtual screen instead of the physical screen. Thereafter, all of the FastWrite family of routines for writing and reading the screen work with the virtual screen. Because all of the objects in **Object Professional**–the ScreenRect, Frame, RawWindow, StackWindow, PackedWindow, and all the higher level objects–build on FastWrite, they work almost as well on the virtual screen as on a real CRT tube! (It's necessary to say "almost" because you can't call the Process method of a high level object and expect people to interact with objects they cannot see.) When you call the Deactivate method, output is again directed to the normal address in video memory.

As we've already described, StackWindows make good use of virtual screens to update partially obscured windows. With these capabilities already built-in, you may never need to declare your own instances of VirtScreen. We've found places where VirtScreen has additional uses: scrolling entry screens are one, and the "rubber band" resize rectangle in the MAKESCRN utility program is another. Perhaps you'll concoct other uses for VirtScreen.

The following example shows some of VirtScreen's capabilities:

```
program VirtScreenExample;  {EXVSCR.PAS}
uses OpRoot, OpString, OpCrt, Opframe, OpWindow;
const
  Rows = 100;
  Cols = 200;
var
  V : VirtScreen;
  R, C, KW : Word;
  S : String;
  Finished : Boolean;
begin
  {Clear the physical screen}
  TextAttr := $70;
  ClrScr;
```

```
  {Allocate a virtual screen}
  if not V.Alloc(Rows, Cols) then begin
    WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
    Halt;
  end;

  {Initialize the virtual screen with an interesting pattern}
  V.Clear($07, ' ');
  V.Activate;
  for R := 1 to Rows do begin
    Str(R:3, S);
    FastWrite(S, R, 1, $07);
    C := (R mod 10)+5;
    while C <= Cols do
      if (C mod 20 = 0) and (C < Cols-5) then begin
        FastWrite(S, R, C+1, $07);
        Inc(C, 5);
      end else begin
        FastWrite(Char(Byte('a')+((C-R-5) mod 26)), R, C, (C-R) and $7F);
        Inc(C);
      end;
  end;
  V.Deactivate;

  {Scroll around the virtual screen}
  R := 1;
  C := 1;
  Finished := False;
  repeat
    V.CopyToScreen(R, C);
    case ReadKeyWord of
      $4800 :                    {Up arrow}
        if R > 1 then
          Dec(R);
      $4B00 :                    {Left Arrow}
        if C > 1 then
          Dec(C);
      $4D00 :                    {Right Arrow}
        Inc(C);
      $5000 :                    {Down arrow}
        Inc(R);
      $011B :                    {Escape}
        Finished := True;
    end;
  until Finished;

  V.Done;
end.
```

The example allocates a virtual screen of 100 rows by 200 columns (40000 bytes). It clears the virtual screen, activates writing to it, uses FastWrite to build an interesting pattern there, and restores writing to the normal screen. Then it enters a keyboard processing loop which copies a portion of the virtual screen to the physical one. By changing the portion that is copied as the arrow keys are pressed it appears that the pattern is scrolling on the screen. Note that we don't limit the values of R and C to stay within the extent of the VirtScreen. The CopyToScreen method is smart enough to clear the remaining portion of the screen with the current values of TextChar and TextAttr (from OPCRT) when the coordinates extend beyond the limits of the virtual screen.

Declaration

```
procedure Activate;
```

Purpose

Activate fast write and read routines on virtual screen.

Description

Activate saves the current values of VirtualSegment, VirtualWidth, and VirtualHeight (from OPCRT) within the object, then sets them to the values for the virtual screen. Until Deactivate is called, almost all techniques for writing to the screen (e.g., FastWrite, RawWindow.wFastWrite) will be redirected to the virtual screen instead. Routines that read from video memory (e.g., FastRead) will also be redirected.

Exceptions include Write and WriteLn, which continue to write to the physical screen. The hardware cursor will also remain visible on the physical screen.

Be careful not to write a string that runs off the lower right hand corner of a virtual screen. When you do this on the physical screen, no particular harm is done. On a virtual screen, however, the overrun wipes out some other portion of the Pascal heap, with possibly disastrous consequences. The FastWrite routines do not protect against this error, although VirtScreen's WriteTo method does.

It is an error to call Activate when the virtual screen has already been activated, but this error is not detected by the VirtScreen object. It is not an error to activate a different VirtScreen when a VirtScreen is already active.

Example

```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
  Halt;
end;
V.Activate;
FastWrite('on the virtual screen', 1, 1, $07);
V.Deactivate;
V.CopyToScreen(1, 1);
```

Allocates and activates a virtual screen, uses FastWrite to write to it, deactivates writing, then copies it onto the physical screen.

See Also

Deactivate WriteTo

Alloc / Clear

Declaration

```
constructor Alloc(Height, Width : Word);
```

Purpose

Create a virtual screen of the specified dimensions.

Description

Allocates a virtual screen that can hold Height rows and Width columns of text. The product 2*Width*Height must be less than 65507 and greater than zero. Alloc allocates 15 more bytes of heap space than the product indicates so that it can choose a virtual video segment within the buffer.

Alloc will fail if the specified dimensions are invalid or if insufficient heap space is available. The cause of an error can be determined by inspecting InitStatus.

Alloc clears the virtual screen using the current values of TextAttr and TextChar from OPCRT. You can initialize it to different values by calling the Clear method.

Example

```
if not V.Alloc(200, 80) then begin
  WriteLn('Alloc error ', InitStatus);
  Halt;
end;
```

Allocates a virtual screen of 200 rows by 80 columns (using 32015 bytes of heap space).

See Also

Init

Declaration

```
procedure Clear(Attr : Byte; Fill : Char);
```

Purpose

Clear virtual screen.

Description

Fills the virtual screen buffer with the Word ((Attr shl 8) or Fill). Note that the Alloc and Init constructors initialize the buffer with Attr=TextAttr and Fill=TextChar.

Example

```
V.Clear($07, ' ');
```

Clears a virtual screen to spaces in dim white on black.

See Also

CopyFromScreen Init

Declaration

```
procedure CopyFromScreen(Row, Col : Word);
```

Purpose

Copy physical screen to position Row,Col on virtual screen.

Description

CopyFromScreen provides a way to initialize at least part of the virtual screen from the physical one. The StackWindow methods make a number of calls to CopyFromScreen(1, 1) for a VirtScreen which is the same size as the physical.

If Row or Col is zero, or if Row or Col exceeds its respective dimension of the virtual screen, CopyFromScreen exits immediately. If Row+ScreenHeight or Col+ScreenWidth exceeds a virtual screen dimension, CopyFromScreen copies what will fit.

Example

```
if not V.Alloc(ScreenHeight, ScreenWidth) then begin
  WriteLn('Alloc error ', InitStatus);
  Halt;
end;
V.CopyFromScreen(1, 1);
```

Allocates and initializes a virtual screen that matches the physical one.

See Also

CopyToScreen

Declaration

```
procedure CopyFromWindow(Row, Col : Word);
```

Purpose

Copy current window to position Row,Col on virtual screen.

Description

Works like CopyFromScreen except that instead of copying the entire physical screen, it copies just the portion within the current OPCRT window coordinates. These coordinates are those set with the last Window procedure call and can be checked by reading OPCRT's WindMin and WindMax variables.

Example

```
if not V.Alloc(ScreenHeight, ScreenWidth) then begin
  WriteLn('Alloc error ', InitStatus);
  Halt;
end;
Window(10, 5, 70, 20);
V.CopyFromWindow(5, 10);
```

Copies the rectangle at (10, 5, 70, 20) to the same relative position within the virtual screen.

See Also

CopyToWindow

CopyToScreen / CopyToWindow

Declaration

```
procedure CopyToScreen(Row, Col : Word);
```

Purpose

Copy from position Row,Col on virtual screen to physical screen.

Description

A rectangle ScreenWidth columns and ScreenHeight rows is copied from position Row,Col on the virtual screen to the physical one.

If Row or Col is zero, CopyToScreen exits immediately. If Row or Col is greater than its respective dimension of the virtual screen, CopyToScreen clears the physical screen using the current values of TextAttr and TextChar. If Row+ScreenHeight or Col+ScreenWidth exceeds a virtual screen dimension, CopyFromScreen copies what is available from the virtual screen and clears the remaining portion of the physical screen using TextAttr and TextChar.

Example

```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
  Halt;
end;
V.Activate;
FastWrite('on the virtual screen', 1, 1, $07);
V.Deactivate;
V.CopyToScreen(1, 1);
```

Allocates and activates a virtual screen, uses FastWrite to write to it, deactivates writing, then copies it onto the physical screen.

See Also

CopyFromScreen

Declaration

```
procedure CopyToWindow(Row, Col : Word);
```

Purpose

Copy from position Row,Col on virtual screen to current window.

Description

Works like CopyToScreen except that instead of copying to the entire physical screen, it copies to just the region within the current OPCRT window coordinates. These coordinates are those set with the last Window procedure call and can be checked by reading OPCRT's WindMin and WindMax variables.

Example

```
if not V.Alloc(ScreenHeight, ScreenWidth) then begin
  WriteLn('Alloc error ', InitStatus);
  Halt;
end;
Window(10, 5, 70, 20);
V.CopyFromWindow(5, 10);
V.CopyToWindow(4, 10);
```

Effectively shifts the specified window's contents one row down by copying to the virtual screen then back to the window using a Row parameter one less.

See Also

CopyFromWindow

Declaration
```
procedure Deactivate;
```

Purpose
Deactivate fast access to virtual screen.

Description
The values of VirtualSegment, VirtualWidth, and VirtualHeight stored when Activate was called are restored. This typically has the effect of redirecting FastWrite output to the physical screen.

It is an error to call Deactivate when the virtual screen has not been activated, but this error is not detected by the VirtScreen object. It is not an error to deactivate a VirtScreen in order to return to another VirtScreen previously activated.

Example
```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
  Halt;
end;
V.Activate;
FastWrite('on the virtual screen', 1, 1, $07);
V.Deactivate;
FastWrite('on the physical screen', 1, 1, $07);
```

Writes one string to the physical screen, calls Deactivate to restore normal writing, then writes to the physical screen.

See Also
Activate

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate a virtual screen.

Description
If the virtual screen buffer was originally allocated from the heap, it is disposed. The internal fields of the VirtScreen are filled with zeros and Root.Done is called.

Example
```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
  Halt;
end;
...
V.Done;
```

Allocates and later deallocates a virtual screen.

See Also
Alloc Init

Init

Declaration

```
constructor Init(Height, Width : Word; BufSeg : Word);
```

Purpose

Initialize a virtual screen with a preallocated buffer.

Description

Use this constructor instead of Alloc if you need complete control over the location of the virtual screen buffer. A good reason for using Init exists when you have alternate text pages on a CGA (or more capable) video adapter and want to use them for virtual screen storage. OPCRT provides the GetPageSegment function which returns the appropriate value to pass for BufSeg.

BufSeg is the segment address of the base of the virtual screen. It is your responsibility to assure that at least 2*Width*Height bytes of memory are available starting at this segment. Init does not attempt to "allocate" this memory, nor does VirtScreen.Done deallocate it.

Init returns an error only if the specified dimensions are invalid.

Init clears the virtual screen using the current values of TextAttr and TextChar from OPCRT. You can initialize it to different values by calling the Clear method.

Example

```
if not V.Init(ScreenHeight, ScreenWidth, GetPageSegment(1)) then begin
  WriteLn('Init error ', InitStatus);
  Halt;
end;
```

Initializes a virtual screen using video page 1 of a color graphics adapter.

See Also

Alloc

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load from stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to VirtScreen.Store. Load reads the size of the virtual screen from the stream. If the screen was originally allocated from the heap (using Alloc), Load allocates a buffer for it again. If the screen was originally initialized by Init, Load uses the exact virtual screen segment that was originally specified.

Note that Load does not read the contents of the virtual screen, nor does it initialize the screen buffer. For VirtScreens, the primary function of Load and Store is to remember the virtual screen's dimensions.

The stream registration procedure for a VirtScreen is VirtScreenStream.

Example

```
uses
  OpCrt, OpRoot, OpFrame, OpWindow;
var
  V : VirtScreen;
  S : BufIdStream;
begin
  {Load V from a stream}
  S.Init('VIRTSCR.STM', SOpen, 1024);
  S.RegisterHier(VirtScreenStream);
  S.Get(V);
  S.Done;

  {Use the VirtScreen}
  V.Clear(TextAttr, ' ');
  ...
end.
```

Instantiates V from the stream created by the Store example using the stream's Get method, which calls Load indirectly.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Store

ReadFrom / ScrollHoriz

Declaration

```
procedure ReadFrom(Number : Byte; Row, Col : Word; var S : string);
```

Purpose

Read number characters from the virtual screen starting at Row,Col.

Description

Calling ReadFrom is almost like performing the following sequence:

```
V.Activate;
FastRead(Number, Row, Col, S);
V.Deactivate;
```

The differences are: ReadFrom will stop reading when it reaches the end of the virtual screen while FastRead won't; ReadFrom may be slightly faster because it knows in advance that it doesn't need to perform snow checking.

Be sure that the string S is long enough to hold Number characters. If it isn't, ReadFrom will probably wipe out memory it shouldn't.

Example

```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to init VirtScreen, InitStatus ', InitStatus); Halt;
end;
V.Activate;
FastWrite('on the virtual screen', 1, 1, $07);
V.Deactivate;
FastWrite('on the physical screen', 1, 1, $07);
V.ReadFrom(30, 1, 1, S);
```

Reads 30 characters from the upper left corner of the virtual screen into the string S. S will receive 'on the virtual screen '.

See Also

FastRead (OPCRT) WriteTo

Declaration

```
procedure ScrollHoriz(Cols : Integer; Attr : Byte; Fill : Char);
```

Purpose

Scroll contents horizontally.

Description

If Cols is zero, ScrollHoriz does nothing; if greater than zero, ScrollHoriz shifts the contents of the virtual screen to the left; if less than zero, to the right. The newly exposed columns of the virtual screen are filled with video attribute Attr and character Fill. If Cols is greater than the virtual screen's width, the whole screen is cleared.

Example

```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to init VirtScreen, InitStatus ', InitStatus); Halt;
end;
V.Clear($07, ' ');
V.WriteTo('on the virtual screen', 1, 1, $07);
V.ScrollHoriz(1, $07, ' ');
```

The upper left corner of the virtual screen contains 'n the virtual screen' after the scroll. Note that none of this appears on the physical screen.

See Also

ScrollVert

Declaration

```
procedure ScrollVert(Rows : Integer; Attr : Byte; Fill : Char);
```

Purpose

Scroll contents vertically.

Description

Works like ScrollHoriz except that scrolling is vertical rather than horizontal. This method is useful if you're using a virtual screen like a normal scrolling text screen by writing rows at the bottom of the screen and scrolling the oldest material off from the top.

Example

```
const
  Height = 200;
var
  Row : Word;
  F : Text;
  S : String[80];
begin
  if not V.Alloc(Height, 80) then begin
    WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
    Halt;
  end;
  V.WriteTo('on the virtual screen', 1, 1, $07);

  {...Open a text file for reading...}

  {Write the text file to the virtual screen just like the TYPE command}
  Row := 0;
  while not Eof(F) do begin
    ReadLn(F, S);
    if Row = Height then
      {Make space for another row}
      V.ScrollVert(1, $07, ' ')
    else
      Inc(Row);
    {Write the string to the current row}
    V.WriteTo(S, Row, 1, $07);
  end;

  {... Do something with the results ...}
end.
```

The example uses a hypothetical text file as a source of strings. It writes these strings starting at row 1 of the virtual screen, incrementing the row by 1 for each string until it reaches the last row. Then it scrolls up by one row and replaces the last row with the new string.

As far as the example goes, none of the results are visible.

See Also

ScrollHoriz

Store / vCols

Declaration
```
procedure Store(var S : IdStream);
```

Purpose
Store to stream.

Description
S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes data describing the virtual screen, including its dimensions and, only if it was initialized by calling Init, its buffer segment. Store does not write the contents of the virtual screen to the stream.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

See BufIdStream (in OPROOT) for proper error checking using streams.

The stream registration procedure for a VirtScreen is VirtScreenStream.

Example
```
uses OpRoot, OpString, OpCrt, Opframe, OpWindow;
var V : VirtScreen;
    S : BufIdStream;
begin
  {Allocate a virtual screen}
  if not V.Alloc(200, 80) then begin
    WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
    Halt;
  end;

  {Write the virtual screen to a stream}
  S.Init('VIRTSCR.STM', SCreate, 1024);
  S.RegisterHier(VirtScreenStream);
  S.Put(V);
  S.Done;
  V.Done;
end.
```
Writes the virtual screen's dimensions and flags to the stream.

See Also
Load

Declaration
```
function vCols : Word;
```

Purpose
Return number of columns in the virtual screen.

Description
vCols returns the number specified as Width when the VirtScreen was instantiated.

Example
```
WriteLn('Virtual screen has ', V.vCols, ' columns');
```
Writes out the number of columns in the virtual screen.

See Also
Alloc vRows

Declaration
```
function vRows : Word;
```

Purpose
Return number of rows in the virtual screen.

Description
vRows returns the number specified as Height when the VirtScreen was instantiated.

Example
```
WriteLn('Virtual screen has ', V.vRows, ' rows');
```
Writes out the number of rows in the virtual screen.

See Also
Alloc vCols

Declaration
```
procedure WriteTo(S : string; Row, Col : Word; Attr : Byte);
```

Purpose
Write a string to virtual screen.

Description
Calling WriteTo is almost like performing the following sequence:
```
V.Activate;
FastWrite(S, Row, Col, Attr);
V.Deactivate;
```
The differences are: WriteTo will stop writing when it reaches the end of the virtual screen while FastWrite won't; WriteTo may be slightly faster because it knows in advance that it doesn't need to perform snow checking.

Example
```
if not V.Alloc(100, 100) then begin
  WriteLn('Unable to Init VirtScreen, InitStatus ', InitStatus);
  Halt;
end;
V.WriteTo('on the virtual screen', 1, 1, $07);
```
Writes the string to the upper left corner of the virtual screen.

See Also
FastWrite (OPCRT) ReadFrom

D. OPPICK - Pick Lists

The OPPICK unit provides only one object type, but the PickList is a whopper of an object! PickList provides a very general way to display a list of arbitrary strings, to allow the user to scroll through the list, and to accept a selection. What distinguishes a PickList from a Menu (described later in the chapter) is dynamic flexibility. The strings displayed by the PickList need not be known in advance; in fact, some applications will read the strings from a disk file on demand or even modify them during the selection process.

Building on this basic idea, the PickList object adds many flexible features that make it useful in diverse situations. Pick lists offer:

- user-defined creation of each item string
- single or multi-column lists
- multiple list orientations
- horizontal and vertical scrolling for large lists
- normal, alternate, or protected (unselectable) item status
- alphanumeric list searching
- mouse support
- specialized headers for "more" and search status
- "flexwrite" coloring for each item
- multiple choice lists

The exact meaning of these terms will become clearer as we present examples and detailed descriptions below. The last two items on this list of features are new to **Object Professional** (as compared to Turbo Professional 5.0).

Because the PickList is derived from the CommandWindow object, it gains all the functionality of that window type. Inherited features relevant to the PickList include:

- adjustable colors, size, and position
- frame decorations such as headers, shadows, and scroll bars
- ability to work in a window stack or on a virtual screen
- CommandProcessor hooks such as keyboard remapping, user exit commands, dynamic exit commands, and background tasks
- context sensitive help hook
- error handling hook

Now that you have some idea of what a PickList object can do, let's start examining the question of how to use one. We'll begin by looking at a simple example program, shown below.

```
program PickListExample;  {EXPICK1.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick;

const
  NumPizzaToppings = 5;       {Number of items on pick list}
var
  PizzaTop : PickList;
  PickWindowOptions : LongInt;

  {$F+}
  procedure PizzaTopping(Item : Word; Mode : pkMode;
                         var IType : pkItemType; var IString : String;
                         PickPtr : PickListPtr);
    {-Sets item string and item options (if any)}
```

```
  begin
    case Item of
      1 : IString := ' Pepperoni';
      2 : IString := ' Sausage';
      3 : IString := ' Mushrooms';
      4 : IString := ' Anchovies';
      5 : IString := ' Onions';
    end;
  end;
  {$F-}

begin
  {Make a PickList with custom window options}
  PickWindowOptions := DefWindowOptions or wBordered;
  if not PizzaTop.InitCustom(35, 5, 45, 10,     {Initial window coordinates}
                             DefaultColorSet,   {ColorSet to use}
                             PickWindowOptions, {Window options}
                             11,                {Column width per item}
                             NumPizzaToppings,  {Number of picklist items}
                             PizzaTopping,      {Item string function}
                             PickVertical,      {Orientation procedure}
                             SingleChoice)      {Command handler}
  then begin
    WriteLn('Failed to Init PickList,  Status = ', InitStatus);
    Halt;
  end;

  {Set some PickList and Frame features}
  PizzaTop.SetSearchMode(PickCharSearch);
  PizzaTop.EnableExplosions(20);
  with PizzaTop.wFrame do begin
    AddShadow(shBR, shOverWrite);
    AddHeader(' PICK ', heTC);
  end;

  {Pick an item}
  PizzaTop.Process;
  PizzaTop.Erase;
  if PizzaTop.GetLastCommand = ccSelect then
    WriteLn('You chose item ', PizzaTop.GetLastChoice,
            ' -  ', PizzaTop.GetLastChoiceString)
  else
    Writeln('No choice was made');
  PizzaTop.Done;
end.
```

The example puts up a PickList showing five pizza toppings. The user can scroll through the list using the arrow keys or jump to a particular item by pressing the first letter of its name. This list accepts a single choice when the <Enter> key is pressed and returns it to the application upon request.

The PizzaTopping procedure demonstrates the most important new concept for using the PickList system. This procedure is an example of what we call the "user-defined string procedure." PickList calls upon this procedure whenever it needs to get the name of one of the list's items, which are numbered from 1 to the total item count. The Item parameter passed to the string procedure specifies which item name is to be returned. There is no restriction on how this procedure actually obtains each item name: it can use a case statement as shown here, refer

to an array of strings, compute the name from some numeric values, or read a disk file. The only constraint is that it should be fast; PickList calls this routine frequently.

Of the various parameters passed to the user-defined string procedure, two of them will always be used. Item specifies the item number to return the string for, and IString is the name that the routine returns. The procedure must always initialize the value of IString. The other parameters are used within more advanced applications; we will return to them later.

The string procedure must be declared correctly in order to compile without error. It must be preceded by the {$F+} directive so that it uses the FAR model; it must not be nested inside of any other procedures or functions; and its parameter list must exactly match the example shown here, whether or not all the parameters are being used.

The example continues with a call to PickList.InitCustom. This constructor's initial parameters parallel those of any other window: the active window coordinates, the colors to use, and the window options. From there the parameters are specific to the PickList. The value 11 specifies the number of screen columns to reserve for each item in the list and NumPizzaToppings (equal to 5) is the total number of items in the list. Note that PickList will place as many whole columns of items across the window as will fit. In this case, the active window width is 11 columns (45-35+1) and the item width is 11 also. If the window width were at least 22 columns, PickList would automatically arrange the items in two columns.

The next parameter, PizzaTopping, specifies the user-defined string procedure. This procedure must meet the requirements already specified.

PickVertical specifies the orientation of the list. The issue is moot for a short, single column list as shown in this example but becomes important for scrolling lists. We'll return to it in a section following.

SingleChoice specifies that this list accepts only a single selection at a time. The alternate value to put here is MultipleChoice. These identifiers are actually the names of procedures interfaced by the OPPICK unit; the constructor calls the named procedure which then initializes the object as appropriate.

If the constructor succeeds, the example goes on to customize the window. It first enables alphanumeric searching with a call to the SetSearchMode method. The PickCharSearch option is one of several provided by OPPICK: PickCharSearch means that when an alphanumeric character is entered, PickList will search to find an item whose name starts with that character. The example also adds shadows and a header.

The call to PizzaTop.Process displays the pick list and allows the user to make a selection. Erase removes the window from the screen. If the user pressed <Enter>, GetLastCommand returns ccSelect, GetLastChoice returns the selected item number, and GetLastChoiceString returns the string associated with the selected item. If the user pressed <Esc>, no selection was made.

Finally, PizzaTop.Done disposes of the PickList and its underlying window.

This example barely scratches the surface of what OPPICK can do. We'll describe more features and provide examples in the following. The MAKESCRN, MAKEMENU, and ENTRY demonstration/utility programs make substantial use of PickLists as well.

Pick List Commands

The OPPICK Process method offers appropriate commands to navigate the list of items and mark selections. The list below arranges them by category. In each case the first line gives the

name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant PickKeySet (in OPPICK.ICD).

Cursor Movement

ccLeft **<Left>, <CtrlS>**
Highlight bar left one item. If the highlight is at the beginning of a row, and wrapping is allowed (more on this later), it is moved to the end of the previous row. If wrapping is not allowed, the highlight does not move.

ccRight **<Right>, <CtrlD>**
Highlight bar right one item.

ccHome **<Home>, <CtrlQ><S>**
Highlight to first item.

ccEnd **<End>, <CtrlQ><D>**
Highlight to last item.

ccUp **<Up>, <CtrlE>, <CtrlW>**
Highlight up one row. If the highlight is at the top of a column, and wrapping is allowed, it is moved to the bottom of the previous column. If it is on the first item, and wrapping is allowed, it is moved to the last item.

ccDown **<Down>, <CtrlX>, <CtrlZ>**
Cursor down one row, with wrapping analogous to ccUp.

ccPageUp **<PgUp>, <CtrlR>**
Scroll display up one page. If the pkAlterPageRow option is in effect, the highlight bar always moves to the top row of the window after ccPageUp. If not, the highlight bar remains on the same window-relative row until the window can scroll no more, at which time the highlight moves to the first row.

ccPageDn **<PgDn>, <CtrlC>**
Scroll display down one page. The pkAlterPageRow option affects this command much like it does ccPageUp.

Multiple Choice Selection

The following commands are interpreted only for multiple choice PickLists, which are activated by passing the MultipleChoice CommandHandler to the PickList's constructor. Selected items are made apparent by using marker strings (check marks or arrows, for example) or a different video attribute.

ccIns **<Ins>**
Mark the current item selected.

ccDel **<Del>, <->**
Deselect the current item.

ccToggle **<Space>, <+>**
If the current item is not selected, select it; otherwise, deselect it.

ccBlkToggle **<CtrlEnter>, <CtrlJ>**
If any items are currently selected, clear all selections. If no items are currently selected, mark all items selected.

ccBlkBegin **<F7>, <CtrlK><B>**
Indicate that the currently highlighted item is the first one in a marked block. The block operation is not complete (and no visible indication of the block appears) until the ccBlkEnd command is given.

ccBlkEnd **<F8>, <CtrlK><K>**
If an item that begins a block has been set with the ccBlkBegin command, and if the current item index is equal to or beyond the begin, then all items from the begin to the current item are marked. The beginning item indicator (stored internally) is reset; another block cannot be marked until the ccBlkBegin command is given again.

ccRestore **<CtrlK><H>**
Deselect all items.

Other

ccSelect **<Enter>**
For a single choice PickList, select the current item and exit the list. For multiple choice, accept the current set of selections and exit the list.

ccMouseSel **<ClickLeft>**
Move the highlight bar to the position indicated by the mouse, if possible. For a single choice list, if the highlight bar is already over the indicated item, select the item and exit the list. For a multiple choice list, if the highlight bar is over the indicated item, toggle its selection. This command can also be used to scroll the pick list by clicking on a scroll bar.

ccQuit **<Esc>, <ClickRight>**
Quit processing the list without making a selection.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling PickCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. The PickList will pass the help procedure the following three parameters: ucPick, @Self, and a help topic number. If pkUseItemForTopic is enabled, the help topic number is the same as the current item's index number. Otherwise, the help topic is the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

The User-Defined String Procedure

As mentioned above, the user-defined string procedure is at the heart of the PickList system. Without it, PickList has no idea what to display. By forcing the programmer to provide this routine, PickList remains as general as possible. If PickList always used an array of strings to get the listed items, many useful applications of PickList would no longer be possible.

Actually, there is more than one way to provide this flexibility. The first we've already described: pass a procedural parameter to the PickList's constructor. The second takes advantage of OOP's virtual methods. When PickList prepares to display an item, it calls the ItemString virtual method, which in turn calls the string routine specified via the procedural parameter. Instead of using the procedural parameter at all, you can create a new object type—derived from PickList—and simply override the ItemString method. This new method can supply each item string directly without referring to any procedure pointers.

The second alternative is more consistent with object-oriented programming methodology. It is even slightly faster because it avoids one level of indirection during each call to get an item string. It's very convenient to create such a derived object, since only one method really needs

to be overridden (although it's usually convenient to provide a specialized constructor method as well).

To make this second alternative clearer, let's rewrite the introductory PizzaTopping program using a virtual method instead of the user procedure.

```
program PizzaListExample;  {EXPICK2.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick;
type
  PizzaList =
    object(PickList)
      constructor Init(X1, Y1, X2, Y2 : Byte);
      procedure ItemString(Item : Word; Mode : pkMode;
                           var IType : pkItemType;
                           var IString : String); virtual;
    end;

constructor PizzaList.Init(X1, Y1, X2, Y2 : Byte);
const NumPizzaToppings = 5;
var   PickWindowOptions : LongInt;
begin
  PickWindowOptions := DefWindowOptions or wBordered;
  if not PickList.InitAbstract(X1, Y1, X2, Y2,
                               DefaultColorSet,   {ColorSet to use}
                               PickWindowOptions, {Window options}
                               11,                {Column width per item}
                               NumPizzaToppings,  {Number of picklist items}
                               PickVertical,      {Orientation procedure}
                               SingleChoice)      {Command handler}
  then
    Fail;
end;

procedure PizzaList.ItemString(Item : Word; Mode : pkMode;
                               var IType : pkItemType;
                               var IString : String);
begin
  case Item of
    1 : IString := ' Pepperoni';
    2 : IString := ' Sausage';
    3 : IString := ' Mushrooms';
    4 : IString := ' Anchovies';
    5 : IString := ' Onions';
  end;
end;

var
  PizzaTop : PizzaList;

begin
  {Make a PizzaList}
  if not PizzaTop.Init(35, 5, 45, 10) then begin
    WriteLn('Failed to Init PizzaList,  Status = ', InitStatus);
    Halt;
  end;

  {Set some PickList and Frame features}
  PizzaTop.SetSearchMode(PickCharSearch);
  PizzaTop.EnableExplosions(20);
```

```
  with PizzaTop.wFrame do begin
    AddShadow(shBR, shOverWrite);
    AddHeader(' PICK ', heTC);
  end;

  {Pick an item}
  PizzaTop.Process;
  PizzaTop.Erase;
  if PizzaTop.GetLastCommand = ccSelect then
    WriteLn('You chose item ', PizzaTop.GetLastChoice,
            ' - ', PizzaTop.GetLastChoiceString)
  else
    Writeln('No choice was made');
  PizzaTop.Done;
end.
```

This time we declare a new object, PizzaList, derived from PickList. Its constructor method encapsulates all the things we know about a PizzaList: there are five items, each 11 characters wide, in a single choice list. Note that the constructor calls one of PickList's constructors, InitAbstract, which is intended specifically for use by derived objects--it requires no procedure parameter because none will be used. The ItemString virtual method does the same thing as the old PizzaTopping procedure did. The main block of the example calls the PizzaList's simplified constructor and after that the program is identical to the original example.

So why do we support the procedure parameter technique at all? There are two reasons. The first is historical: Turbo Professional's pick lists used this technique and there is value in maintaining a degree of conceptual continuity. The second is more important: by using procedure parameters a single PickList object can display lists whose format varies dynamically. A good example is the DirList object, which displays DOS directories. The format for each filename must be very flexible--with or without date and time, with or without file size, etc.--and it must be changeable while the program runs. Of course, this flexibility could be achieved by writing a complex ItemString method with lots of flags and options. This approach would waste both time and code space. Using procedure parameters, only the formats that are actually used are linked in and execution can jump immediately to the procedure whose format is in use at the moment.

You can make your own choice between the two approaches. Whenever it makes sense, however, we recommend that you take the OOP approach and derive your own specialized PickList types. We've done that in many of the **Object Professional** demonstration programs: the Calendar and PhoneBook objects in DESKPOP, as well as the frame, header, and character lists in MAKEMENU and MAKESCRN.

With that general issue covered, we can now provide more details about how to use a string procedure or virtual method.

Each user-defined string procedure must be compiled under the FAR model, must be global, and must match the pkStringProc type declaration, which means that it has exactly the same parameters as the following:

```
{$F+}
procedure MyStringProc(Item : Word;
                       Mode : pkMode;
                       var IType : pkItemType;
                       var IString : String;
                       PickPtr : PickListPtr);
```

The corresponding virtual method has a closely related declaration:

```
procedure ItemString(Item : Word;
                     Mode : pkMode;
                     var IType : pkItemType;
                     var IString : String); virtual;
```

Although the virtual method appears to lack the final PickPtr parameter, it really has one: every method is passed the address of Self as the last (hidden) parameter. PickPtr serves the same purpose for MyStringProc, allowing the procedure to refer to all the methods and data fields of the PickList instance for which it is building the item name. For example, if MyStringProc needed to call the GetDefaultChoice method, it would use the following syntax:

```
InitChoice := PickPtr^.GetDefaultChoice;
```

The other parameters are interpreted identically in either case. Item is the sequential number of the item whose string the procedure must return. The first item is number 1 and the other ones increase sequentially to the total count.

Mode specifies why the PickList is calling the string procedure. Depending on which of the following values Mode takes, the routine may be able to skip certain processing steps.

pkDisplay
: The string returned will be used for display. Padding characters and control characters used to activate flexwrite formatting should be part of the string.

pkSearch
: The string returned will be used for alphanumeric searching. Padding and control characters should not be added to the string.

pkGetType
: PickList isn't looking for a string at all. It just wants to know whether the specified item is normal or protected from selection. To save time the string procedure can initialize just the IType parameter and exit immediately.

IType returns the category of the item, which is one of the following: pkNormal, pkAlternate, pkProtected, or pkPrivate. PickList initializes IType to pkNormal before calling the string procedure, so you don't need to make an assignment to it unless the item isn't normal. pkAlternate items are treated just like pkNormal items except that they use a different set of video attributes. A good example of a pkAlternate item is a subdirectory name in a directory listing. pkProtected items are displayed in yet another set of attributes; in addition, the highlight bar will never move to one of these items. They can be used to provide titles or white space within a PickList. pkPrivate is a category reserved for TurboPower; you should not set IType to this value.

IString is the string that returns the name of the specified item. Except when Mode is pkGetType, you must always provide a value for IString. OPPICK doesn't care where you get the value of the string, so you can let your creativity loose. Just remember that this routine is called frequently, so you don't want to do anything that takes too much time.

OPPICK truncates or pads the returned string if its visible length (after removing flexwrite control characters) doesn't match the specified column width. OPPICK will also add padding and selection markers to the returned string if appropriate options are enabled.

You might wonder why the string procedure isn't a string *function*, since its primary purpose is to return a string. By passing the string as a var parameter rather than a function result, stack usage is minimized, especially when the string is transferred among several nested routines. And there are at least three calls nested during typical use: the PickList display routine calls the

CompleteItemString virtual method (which returns the padded, marked string), which in turn calls the ItemString virtual method (which returns the unpadded string), which in turn calls the user-defined string procedure. The var parameter approach saves at least 3*256 bytes of stack space, and probably quite a bit more.

OPPICK makes no guarantees about the sequence or number of calls to the string procedure. Some people have assumed (while using Turbo Professional) that the string procedure is called in increasing sequential order for each item and that the last call per screen update is for the highlighted item. This is not necessarily the case. In fact, OPPICK takes special pains to minimize the number of calls to the string procedure since it is often the primary performance bottleneck. For example, when the pick list must scroll up by one row, OPPICK performs the scrolling by direct video moves and calls the string procedure only to display the row that newly appears.

Pick List Orientations

When the list items are arranged in more than one column, OPPICK can display the items in any of three different orientations. It's easier to demonstrate these orientations visually than to describe them. Assume that we have an 11 item list (the letters A through K) arranged in three columns in a window three rows tall. Here's what we would see for each of the three orientations when the list was first displayed:

```
PickVertical      PickHorizontal      PickSnaking
 A   E   I         A   B   C          A   D   G
 B   F   J         D   E   F          B   E   H
 C   G   K         G   H   I          C   F   I
```

For PickVertical, increasing item numbers are arranged in vertical columns. Note that items D and H are below the window and will not appear until the window scrolls upward.

For PickHorizontal, increasing items are arranged in horizontal rows. The right hand edge of one row is continued immediately at the left edge of the next row. Scrolling the example up by one row would show the two remaining elements, J and K.

PickSnaking is like PickVertical in that the items are arranged in columns. However, it is like PickHorizontal in that the top of one column picks up where the bottom of the previous column left off. Scrolling the example window left by one column would make items J and K appear in the right hand column.

Which orientation is best will depend on the application. For sorted directory lists, we prefer the PickSnaking orientation because the eye naturally tends to scan down the columns and yet, unlike PickVertical, no breaks in the sequence occur.

Note that the list orientation has no effect on the use of the cursor keys within the list. Pressing the <Down> arrow key, for example, will always move the highlight bar down one row.

It's easy to use any of the three orientations. Just specify the orientation name–PickVertical, PickHorizontal, or PickSnaking–in the Orientation parameter of the PickList constructor. To change the introductory example to the PickHorizontal orientation, we would modify only the PizzaTop constructor call:

```
if not PizzaTop.InitCustom(35, 5, 45, 10,      {Initial window coordinates}
                           DefaultColorSet,    {ColorSet to use}
                           PickWindowOptions,  {Window options}
                           11,                 {Column width per item}
                           NumPizzaToppings,   {Number of picklist items}
                           PizzaTopping,       {Item string function}
{ ------> }                PickHorizontal,     {Orientation procedure}
                           SingleChoice)       {Command handler}
then begin
  ...
```

You can change the orientation whenever the PickList is not active by calling ChangeOrientation. Or check the current orientation at any time by calling GetOrientation.

Note that the three orientations are actually the names of procedures interfaced by OPPICK. These procedures, when called by the PickList, perform the initialization needed to make the cursor keys and the display logic work. Each additional orientation you use in a program causes Turbo Pascal to link in more executable code.

Window Sizing for Pick Lists

OPPICK provides several other techniques besides the three orientations to control where specific items are displayed within a list and how the pick window fits around them. We'll summarize them here.

One of the PickList option flags is called pkMinHeight. When this option is active, as it is by default, all PickList constructors will automatically reduce the height of the pick window if the list's items do not use every row of the window. For example, suppose that a list contains a single column of 7 items total. If the constructor's minimum and maximum Y coordinates were 5 and 20, respectively, that would leave room for 16 items. The constructor would reduce the maximum Y coordinate to 11 before calling the CommandWindow constructor, thus squeezing out the empty rows. Because this adjustment occurs before the window is instantiated, there is no resulting heap fragmentation.

The PickList method SetRowLimits affects the minimum and maximum number of items in each column of the list. Suppose we take the A-K example from the previous orientation discussion, using PickVertical and a minimum row limit of 5. What we'd see initially would be

```
A   F   K
B   G
C   H
```

- Note that the items D and E are below the edge of the window, as are I and J. The minimum row limit is forcing 5 items into each column.

Similarly, the maximum row limit specifies the maximum number of items in each column. Unlike the pkMinHeight option just described, the maximum row limit does not affect the height of the pick window. It controls just the arrangement of the items within the window. Going back to the A-K example, if we set the maximum row limit to 2 (without changing anything else), we'd see

```
A   E   I
B   F   J
-   -   -
```

The underscores indicate empty regions within the window.

Another method, OptimizeSize, completely reconfigures the pick window when the number of items, their width, or the orientation has changed. We use this method in DESKPOP's file manager window whenever the display mode is changed from compact to full-information. The method adjusts the window coordinates to approach the originally requested size by maximizing the number of item columns, minimizing the unused width and height of the window, and simultaneously staying within the RawWindow window limits. We'll provide more details in the OptimizeSize method description. Before reading it you should experiment with DESKPOP's file manager window to see OptimizeSize in action.

Multiple Choice Lists

One of the most popular enhancements requested for Turbo Professional's pick list unit was the ability to select and mark more than one item at a time. While this capability was easily implemented by making good use of the user-defined string function, many people seemed to expect that the unit would provide it automatically. As a result, we've done that with OPPICK.

Multiple choice pick lists are useful for many applications. We've implemented one common one for **Object Professional**: the file manager window in DESKPOP. This window can apply a specified file operation--deletion, renaming, copying--to an entire list of marked files. Another common need is to select a set of options, perhaps for an installation program.

The PickList object works in concert with another **Object Professional** object type, the BitSet, to provide multiple choice capability. Each bit in the BitSet corresponds to one item in the list; when the item is marked, the bit is set, and vice versa. Because BitSets are so efficient--they pack 8 bits in each byte--there is very little data space penalty associated with multiple choice lists. Allocation and management of the BitSet is automatic and the PickList provides a group of methods that make it easy to determine which items are selected at any time.

OPPICK also provides methods to visually identify which items are selected. The SetSelectMarker method specifies string to display to the left and right of each selected item. Typically these strings will use the "check mark" character (#251) or some form of arrow character (e.g., #16) to indicate a selected item.

When the multiple choice option is active, OPPICK activates new commands in the pick window. These include commands for marking blocks of items, toggling the selection state, and clearing the whole list. See the section "Pick List Commands" above for more information.

To activate multiple choice operation, you simply specify MultipleChoice for the CommandHandler parameter to any PickList constructor.

Here's an example, again based on PizzaToppings, of a multiple choice PickList. You'll see that very few changes are required.

```
program PizzaListExample;  {EXPICK3.PAS}
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick;
type
  MultiPizzaList =
    object(PickList)
      constructor Init(X1, Y1, X2, Y2 : Byte);
      procedure ItemString(Item : Word; Mode : pkMode;
                           var IType : pkItemType;
                           var IString : String); virtual;
    end;

constructor MultiPizzaList.Init(X1, Y1, X2, Y2 : Byte);
const NumPizzaToppings = 5;
var   PickWindowOptions : LongInt;
```

```
begin
  PickWindowOptions := DefWindowOptions or wBordered;
  if not PickList.InitAbstract(X1, Y1, X2, Y2,
                               DefaultColorSet,   {ColorSet to use}
                               PickWindowOptions, {Window options}
                               11,                {Column width per item}
                               NumPizzaToppings,  {Number of picklist items}
                               PickVertical,      {Orientation procedure}
    {--------> }              MultipleChoice)    {Command handler}
  then
    Fail;
  SetSelectMarker(#251, '');
end;

procedure MultiPizzaList.ItemString(Item : Word; Mode : pkMode;
                                    var IType : pkItemType;
                                    var IString : String);
begin
  case Item of
    1 : IString := 'Pepperoni';
    2 : IString := 'Sausage';
    3 : IString := 'Mushrooms';
    4 : IString := 'Anchovies';
    5 : IString := 'Onions';
  end;
end;

var
  PizzaTop : MultiPizzaList;
  SelectedItem : Word;

begin
  {Make a MultiPizzaList}
  if not PizzaTop.Init(35, 5, 45, 9) then begin
    WriteLn('Failed to Init MultiPizzaList,  Status = ', InitStatus);
    Halt;
  end;

  {Pick an item}
  PizzaTop.Process;
  PizzaTop.Erase;
  if PizzaTop.GetLastCommand = ccSelect then
    if PizzaTop.GetSelectedCount > 0 then begin
      WriteLn('You selected:');
      PizzaTop.InitSequence(SelectedItem);
      while PizzaTop.HaveSelected(SelectedItem) do begin
        WriteLn(PizzaTop.GetItemString(SelectedItem));
        PizzaTop.NextSelected(SelectedItem);
      end;
    end else
      Writeln('No selections');
  PizzaTop.Done;
end.
```

Note that the call to PickList.InitAbstract specifies the MultipleChoice option. The call to SetSelectMarker specifies that a check mark at the left edge will identify a selected item. The leading space in the string for each topping is now removed; its place is now taken by the one character select marker.

After the call to PizzaTop.Erase we now check to see whether any items were selected. If any were selected, the series of calls to InitSequence, HaveSelected, and NextSelected scans through all of the selected items. We write out the name of each one by calling GetItemString.

There may be times when you want to preselect some of the items. This is easy, but it requires that you know one more fact: PickList doesn't allocate its BitSet until it is forced to because the number of items may change after the constructor is called. (This often happens for directory lists.) Hence, to preselect some items you must explicitly allocate the BitSet and then mark the desired items. Try adding the following lines to the example, at the end of MultiPizzaList.Init:

```
AllocateSelectSet(NumPizzaToppings);
SelectItem(2);
SelectItem(5);
```

Allocating the BitSet also clears it. You'll now see that items 2 (Sausage) and 5 (Onions) are selected when the list is first displayed.

PickList has other methods for managing multiple choice lists. Be sure to read about the following methods:

```
function GetItemString(Item : Word) : string;
  {-Return specified item string in search format}
procedure SetSelectMarker(Left, Right : String);
  {-Define strings to mark selected items}
procedure AllocateSelectSet(MaxItems : Word);
  {-Allocate the select set for the maximum items}
procedure ClearSelected;
  {-Clear selected items}
procedure MarkAllSelected;
  {-Mark all items selected}
procedure SelectItem(Item : Word); virtual;
  {-Select an item}
procedure DeselectItem(Item : Word); virtual;
  {-Deselect an item}
function ItemIsSelected(Item : Word) : Boolean;
  {-Return True if item is selected}
function GetSelectedCount : Word;
  {-Return number of items selected}
procedure InitSequence(var Item : Word);
  {-Prepare to get first selected item}
function HaveSelected(Item : Word) : Boolean;
  {-Return True if there is a selected item}
procedure NextSelected(var Item : Word);
  {-Position at the next selected item}
```

String Searching

Common behavior for menu systems is to select an item when the first character of the item name is typed. The PickList supports this behavior and generalizes it as well.

OPPICK's Process method calls the virtual method ItemSearch whenever it detects the ccChar command. ItemSearch calls a user-defined search function which must match the following prototype definition:

```
{$F+}
function PickSearch(P : PickListPtr) : Boolean;
```

The function can refer to the data fields and methods of the PickList object in order to determine whether one of the items matches. If there is a match, the user-defined function should set the current pick item accordingly. If the Process method should exit as a result of the match, the user-defined function should return True. If there is no match or if Process should remain active anyway, the function should return False.

OPPICK provides several of these functions already, so you probably won't need to write one yourself. Here are the predefined ones:

```
function PickNoSearch(P : PickListPtr) : Boolean;
  {-No searching mode}
function PickStringSearch(P : PickListPtr) : Boolean;
  {-String searching mode}
function PickAltStringSearch(P : PickListPtr) : Boolean;
  {-Alternate string searching mode}
function PickCharSearch(P : PickListPtr) : Boolean;
  {-Character searching mode}
function PickCharExit(P : PickListPtr) : Boolean;
  {-Character searching with exit on match}
function PickAnyCharExit(P : PickListPtr) : Boolean;
  {-Exit on any alpha character. No highlight repositioning}
```

A newly constructed PickList uses the PickNoSearch function; alphanumeric characters are ignored in this mode. Select an alternate searching mode by calling the SetSearchMode method with the name of the string function as a parameter, e.g.:

```
PizzaTop.SetSearchMode(PickStringSearch);
```

PickStringSearch performs an incremental search by appending each character that produces a match to an internal string. The capacity of this string is determined by the OPPICK constant pkMaxSearchLen, which defaults to 15. Searching starts with the currently highlighted item. Each time a further match is obtained, OPPICK advances the current item to that item. The Process method never exits as a result of a match (that is, the user must press <Enter> to complete the selection). When a non-matching character is entered, the internal search string is cleared and the current item remains unchanged.

PickAltStringSearch differs from PickStringSearch only after detecting a non-matching character. In that case, the internal search string is initialized to the last character entered (the non-matching character) and another search attempted. If this second attempt produces a match, the new matching item is highlighted. Otherwise the search string is cleared.

PickCharSearch searches for only a single character at a time. The search always starts one item beyond the current one; in this way, pressing the same character repeatedly will advance the highlight through all of the items starting with the same character. The user must press <Enter> to complete a selection.

PickCharExit searches for a single character, starting with the current item. If it finds a match, it highlights the matching item, changes the last entered command from ccChar to ccSelect, and returns True to force Process to exit.

PickAnyCharExit doesn't do any matching at all. It simply forces Process to exit when any alphanumeric character is entered. GetLastCommand will return ccChar upon return from Process. Applications for this search mode are left to your imagination.

The internal search engine uses a similar strategy for all of the search modes. Protected items are never matched. Searching is not case sensitive. Each item string is generated by calling the CompleteItemString method using the pkSearch mode (which means that flexwrite control characters and selection markers should not be part of the returned string). Leading white space is trimmed from the returned string. Characters to the left of the starting search column (set with the SetSearchStart method) are deleted. Finally, if Pos(SearchString, AdjustedItemString) equals 1, there is a match. If not, the search proceeds with the next item, wrapping to the start of the list when it reaches the end. The search loop continues until it reaches the initial item again.

As with the user-defined string procedure, the search function can be implemented either by overriding the ItemSearch method in a derived type or by calling SetSearchMode to assign a different function pointer. See the preceding section "The User-Defined String Procedure" for arguments in favor of each approach.

Mouse Support

Mouse support is built into the PickList object; all you need to do is activate it. To try it, add OPMOUSE to the Uses statement of the PizzaTopping example, then add the following lines after the call to PizzaTop.Init:

```
if MouseInstalled then begin
  with DefaultColorSet do
    SoftMouseCursor($0000, (ColorMono(MouseColor, MouseMono) shl 8)+$04);
  PickCommands.cpOptionsOn(cpEnableMouse);
  PizzaTop.wFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
end;
```

This code first assures that a mouse is installed. If so, it defines the mouse cursor characteristics. This step is optional since a software mouse cursor is the default, but you'll probably want to set its colors to be consistent with the rest of your ColorSet. You should always use a software (as opposed to hardware) mouse cursor with the PickList object. See OPMOUSE for more information about defining the mouse cursor.

The call to PickCommands.cpOptionsOn initializes the PickList's command processor to support the mouse. See OPCMD for more information, especially if you are using background task hooks.

Finally this code adds an optional scroll bar. You can select pick items using a mouse even without having a scroll bar, and you can have a scroll bar without using the mouse. Note that the minimum (0) and maximum (MaxLongInt) user limits are arbitrary at this point; the PickList updates them to be consistent with the actual configuration of the pick list. This example uses only a vertical scroll bar on the right edge of the frame. PickList also supports horizontal scroll bars. The scroll bars act as you expect them to: the slider moves as the current item is changed; clicking on the bar changes the highlighted item; clicking on the increment/decrement arrows scrolls the list. If the PickList's pkMousePage option is enabled, the arrows scroll a full page at a time; otherwise, a row (or column) at a time.

Specialized Frame Headers

Two specialized header types provide visual enhancement of pick lists. The first is called a "more header." This is typically used with scrolling pick lists to indicate that more items are available outside of the window. It is the non-mouse-oriented alternative to a scroll bar. Add a more header to the window's main frame by calling AddMoreHeader:

```
procedure AddMoreHeader(PromptStr : String80;
                        Posn : HeaderPosType;
                        UpStr, DnStr, SepStr : String10;
                        UpPos, DnPos, SepPos : Byte);
```

As you can see, this provides considerable flexibility for specifying the header's position and contents. A common call to AddMoreHeader would read:

```
PizzaTop.AddMoreHeader('|| for more', heBR, #24, #25, '', 1, 2, 0);
```

PickList replaces the first two characters of PromptStr with the up (#24) and down (#25) arrows depending on what portion of the list is visible and then updates the header display. We'll describe all the parameters in detail on AddMoreHeader's documentation page.

The second specialized header is a search header. Use this header when the PickList's search mode has been set to PickStringSearch or PickAltStringSearch and you want a visual display of the incremental search string. Add a search header with the following method:

```
procedure AddSearchHeader(MaxLen : Byte; Posn : HeaderPosType);
```

MaxLen specifies the maximum number of characters of the search string to display; additional ones are truncated, usually to keep the header from overflowing the frame or from overwriting another header. There's no point in making MaxLen larger than pkMaxSearchLen (15), since that's the longest search string OPPICK will ever store.

Using FlexWrite for Item Display

The items in the example PickLists we've shown so far have been displayed in a uniform color. OPPICK also has the capability to use multiple video attributes within each pick item, by using the FlexWrite routine from OPCRT. The most common use for this technique is to highlight the first character in each item, indicating to the user that typing this character will select the item. Let's modify the PizzaTopping example once more to demonstrate this technique.

```
program PizzaListExample;  {EXPICK4.PAS}
uses OpCrt, OpMouse, OpRoot, OpCmd, OpFrame, OpWindow, OpPick;
type PizzaList =
       object(PickList)
         constructor Init(X1, Y1, X2, Y2 : Byte);
         procedure ItemString(Item : Word; Mode : pkMode;
                              var IType : pkItemType;
                              var IString : String); virtual;
       end;

constructor PizzaList.Init(X1, Y1, X2, Y2 : Byte);
const NumPizzaToppings = 5;
      ColorFlex : FlexAttrs = ($07, $0F, $01, $70);
      MonoFlex : FlexAttrs = ($07, $0F, $01, $70);
var   PickWindowOptions : LongInt;
begin
  PickWindowOptions := DefWindowOptions or wBordered;
  if not PickList.InitAbstract(X1, Y1, X2, Y2,
                               DefaultColorSet,   {ColorSet to use}
                               PickWindowOptions, {Window options}
                               11,                 {Column width per item}
                               NumPizzaToppings,  {Number of picklist items}
                               PickVertical,      {Orientation procedure}
                               SingleChoice)      {Command handler}
  then
    Fail;
```

```
  SetSearchMode(PickCharSearch);
  {Enable flexwriting}
  SetPickFlex(pkNormal, False, ColorFlex, MonoFlex);
end;

procedure PizzaList.ItemString(Item : Word; Mode : pkMode;
                               var IType : pkItemType;
                               var IString : String);
begin
  case Item of
    1 : IString := ' Pepperoni';
    2 : IString := ' Sausage';
    3 : IString := ' Mushrooms';
    4 : IString := ' Anchovies';
    5 : IString := ' Onions';
  end;
  if Mode = pkDisplay then begin
    {Insert flexwriting control characters}
    Insert(^A, IString, 3);
    Insert(^A, IString, 2);
  end;
end;

var
  PizzaTop : PizzaList;

begin
  {Make a PizzaList}
  if not PizzaTop.Init(35, 5, 45, 10) then begin
    WriteLn('Failed to Init PizzaList,  Status = ', InitStatus);
    Halt;
  end;

  {Pick an item}
  PizzaTop.Process;
  PizzaTop.Erase;
  if PizzaTop.GetLastCommand = ccSelect then
    WriteLn('You chose item ', PizzaTop.GetLastChoice,
            ' -  ', PizzaTop.GetLastChoiceString)
  else
    Writeln('No choice was made');
  PizzaTop.Done;
end.
```

The SetPickFlex call in Init enables flexwriting for this pick list. For this case we'll flexwrite only normal items (pkNormal) that aren't the currently selected (False in the second parameter). We specify two arrays of video attributes, one used for color video modes and the other for monochrome. These arrays are interpreted the same way as for calls to FlexWrite, described in OPCRT. You can build the attribute arrays from specific elements of a ColorSet, or you can specify them explicitly as we've done here.

Init also enables character searching so that the highlighted characters will have some meaning.

The ItemString procedure must be modified to support flexwriting. When Mode is set to pkDisplay (and only then) we insert two additional control characters into each item string. The first <CtrlA> tells FlexWrite to start writing with its first alternate attribute and the second tells it to stop. The effect in this example is to write the first non-blank character of each item in bright video (attribute $0F).

You can create many additional special effects by using flexwriting. Each flex array has three attributes besides the base value: these are activated with <CtrlA>, <CtrlB>, and <CtrlC> characters embedded in each item string. See OPCRT.FlexWrite for more information. SetPickFlex allows flexwriting for normal, alternate, and protected items in their unselected and selected states. When all the combinations are tallied, PickList must store 64 different video attributes; certainly enough for the most colorful application!

Protected Items

Pick lists can contain items that are never meant to be selected. Such items might be used as sub-titles, white space, or separator bars that scroll along with the list itself.

Such items are created by the user-defined string procedure. The procedure returns the item string as usual, but it must set the IType parameter to pkProtected. The PickList then knows two things: to draw the string in a different attribute (ProItemColor or ProItemMono from the ColorSet) and to keep the highlight bar from landing on the protected item.

In our PizzaTopping example, if we wanted to protect 'Onions' from selection, we'd just add the following line of code at the end of the string procedure:

```
if Item = 5 then
  IType := pkProtected;
```

When many items are protected, it may be worthwhile to instantiate an auxiliary BitSet to keep track of which items are protected. Then you'd add something like the following to the string procedure:

```
if ProtectedItems.BitIsSet(Item) then
  IType := pkProtected;
```

It's worth describing how OPPICK manages cursor movement in the presence of protected items. First, note that it's a programming error to protect every item in the list. If you call Process for such a list, OPPICK won't highlight any items at all and will simply wait until <Esc> is pressed. In a situation where you cannot guarantee that at least one item is unprotected, you should assure that PickList's NumSelectableItems function returns a non-zero value before calling Process.

If the user presses an arrow key while processing the list and the resulting item is protected, OPPICK enters a special search loop. It repeatedly executes the just-processed command until a) it reaches an unprotected item, b) the item doesn't change (because it's stuck at the bottom of a column, for example), or c) it reaches the original item again. If no unprotected item is found by this loop, OPPICK exhaustively searches the item list (in a direction consistent with the last command) to find any unprotected item.

You can design degenerate pick lists where this heuristic behavior doesn't seem to make much sense; even so, it gives logical results in almost all cases. One seemingly illogical case occurs when the last item in a list is protected and the user mouse-clicks on the last position of the list's scroll bar. OPPICK attempts to select the last item and fails; repeating the downward action leads it to the first item on the list.

If you attempt to select a protected item with the mouse, OPPICK simply ignores the attempt. Similarly, string searching ignores protected items.

The User Move Hook

Sometimes you may want to perform a specified action whenever the user changes the highlighted pick item. A common example is to display a help line or detailed description of

each item. The "user move hook" calls a specified procedure just before each pick command is entered so that you can perform such tasks.

Like PickList's other hooks, you can access this one either by overriding a virtual method or by using a procedure pointer. The method to override is

```
procedure PreMove; virtual;
  {-Called just prior to getting each keyboard command}
```

PickList's implementation of PreMove calls the procedure specified by

```
procedure SetMoveProc(MP : pkGenlProc);
  {-Specify user move procedure}
```

The default move procedure does nothing. An alternate move procedure must be compiled with the FAR model using the {$F+} directive, it must be global, and it must match the following prototype declaration:

```
{$F+}
procedure MyMoveProc(P : PickListPtr);
```

The move procedure can access any of the PickList's data fields or methods by dereferencing the pointer passed to it. Probably the most common need is to get the currently selected item number; this is done with a statement like the following:

```
CurrentItemNumber := P^.GetLastChoice;
```

The move procedure can perform any desired action, with the following restrictions: the active window must be the same on exit as it was on entry and the selected pick item cannot be changed.

At the time PreMove is called, the pick list's internal variables and display have been fully updated to account for the previous command. The mouse cursor, if any, is hidden. The next command has not yet been obtained from the keyboard. There is no guarantee that the currently selected item is different than it was during the previous call to PreMove, although that will usually be the case.

Here's an example of a move procedure that goes with our PizzaTopping example:

```
{$F+}
procedure ToppingSuggestions(P : PickListPtr);
var
  S : String[39];
begin
  case P^.GetLastChoice of
    1 : S := 'A required topping';
    2 : S := 'Optional          ';
    3 : S := 'A little is OK    ';
    4 : S := 'If you must...    ';
    5 : S := 'Not recommended   ';
  end;
  FastWrite(S, ScreenHeight, 1, TextAttr);
end;
{$F-}
...
PizzaTop.SetMoveProc(ToppingSuggestions);
...
```

The move procedure writes more information about each pizza topping to the last row of the screen. The routine is activated by calling SetMoveProc after PizzaTop is instantiated.

Constants

```
pkStick           = $0001;     {Stick at edges of scrolling lists}
pkExitAtEdges     = $0002;     {Exit picklist for cursor at extreme edges}
pkAlterPageRow    = $0004;     {Alter screen row when paging}
pkMousePage       = $0008;     {Clicking on scroll bar scrolls by page}
pkDrawActive      = $0010;     {Draw and Pick leave selected item visible
                                and frame status headers showing}
pkMinHeight       = $0020;     {Minimize height depending on number of items}
pkSetDefault      = $0040;     {Store the default choice when entering Pick}
pkOptFullExtent   = $0080;     {When optimizing size, account for full
                                extent of window, including shadows}
pkUseItemForTopic = $0100;     {Use the current item number for help topic,
                                else use the code in the command window}
```

These are the options that affect the behavior of a PickList. pkStick forces the highlight bar to "stick" at the edges of scrolling lists. If this option is not enabled or the list fits completely in the window, the bar wraps to the next row or column.

pkExitAtEdges also controls behavior at the edges of a list: if enabled, the Process method exits when a cursor key pointing outward from the window is pressed. GetLastCommand will return one of the four special values ccExitAtTop, ccExitAtBot, ccExitLeft, or ccExitRight. This option is appropriate only when a non-scrolling pick list is embedded in a larger structure such as an EntryScreen.

pkAlterPageRow affects the behavior of <PgUp> and <PgDn> (see "Pick List Commands").

When pkMousePage is enabled, clicking on the decrement or increment arrow of a vertical scroll bar is treated like pressing <PgUp> or <PgDn>. Otherwise, it is treated like pressing <Up> or <Down>.

If pkDrawActive is in effect, the current item is highlighted and the specialized pick headers are visible after the Process method exits. Otherwise the highlight and headers are erased.

When pkMinHeight is set prior to calling a PickList constructor, the pick window height will be reduced until the specified number of items completely fills the rows of the window.

pkSetDefault indicates that Process should store the initially selected item number in an internal data field immediately upon entry. The data field can be accessed with the GetDefaultChoice method. The user string procedure can refer to this value, perhaps in order to highlight the associated item. If pkSetDefault is not enabled, the internal data field is changed only by calling SetDefaultChoice.

When pkOptFullExtent is enabled, the OptimizeSize method takes into account the pick window's shadows while adjusting the window size. The window won't be allowed to grow so much that the shadows must be clipped.

If pkUseItemForTopic is set, Process passes the current item number as the topic number to the user help routine. Otherwise, the help topic is the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

```
DefPickOptions : Word = pkStick+pkAlterPageRow+pkDrawActive+
                        pkMinHeight+pkSetDefault+pkOptFullExtent;
```

The default options for a PickList.

```
BadPickOptions : Word = pkItemWrap+pkPickActive+pkBitSetAlloc+pkRedrawPage+
                        pkRedrawOne+pkFlexWrite+pkMultiChoice;
```

Options used internally by PickList. The pkOptionsOn and pkOptionsOff methods will not change these bits of a PickList option word.

```
PickKeyMax = 160;
PickKeyID  : string[11] = 'oppick keys';
PickKeySet : array[0..PickKeyMax] of Byte = (...);
PickCfgEnd : Byte = 0;
```

PickKeyId is an ID string used to mark the beginning of the configuration data area for OPPICK; PickCfgEnd marks the end of the configuration data area. In between them is PickKeySet, the command table used by PickCommands. PickKeyMax is the last valid index into the table.

```
pkMaxSearchLen = 15;
```

Maximum length of string used during incremental searching. Increasing this value increases the size of a PickList instance variable. The size matters only when PickStringSearch or PickAltStringSearch is specified as the search mode.

```
pkNormal    = 0;          {Normal items}
pkAlternate = 1;          {Alternate items}
pkProtected = 2;          {Protected items}
pkPrivate   = 3;          {Reserved for TurboPower}
```

Categories of items in a pick list. See the introduction to this section for more information.

```
pkNoOrient   = 0;
pkVertical   = 1;
pkHorizontal = 2;
pkSnaking    = 3;
```

Values returned by the GetOrientation method to indicate the orientation of the PickList.

Types

```
String10 = String[10];
String80 = String[80];
```

String types used while specifying a "more header."

```
pkItemType       = pkNormal..pkPrivate;
```

Valid item categories (see Constants, above).

```
pkBaseColorArray = array[pkItemType, Boolean] of FlexAttrs; {unsel/selected}
pkColorArray     = array[Boolean] of pkBaseColorArray; {mono/color}
```

Arrays used for video attribute storage in a PickList. Note that there are 64 independent color values. If flexwriting is not enabled, at most 16 of these are used.

```
pkSearchString   = String[pkMaxSearchLen];
```

Data type used to hold the current incremental search string.

```
pkMode           = (pkDisplay, pkSearch, pkGetType);
```

Modes that may be passed to the user string procedure. See the introductory discussion.

```
pkStringProc     = procedure (Item : Word; Mode : pkMode;
                              var IType : pkItemType; var IString : String;
                              PickPtr : PickListPtr);
```

Procedural type that defines parameters of a user string procedure.

```
pkGenlProc       = procedure (P : PickListPtr);
```

Procedural type that defines the parameters of a user move procedure as well as other procedure variables used internally by the PickList.

```
pkSearchFunc     = function (P : PickListPtr) : Boolean;
```

Procedural type that defines the parameters of a user search function.

```
MoreRec =
  record
    ...
  end;
```

Record type that stores the details of a specialized "more header." This type is also used by OPHELP for a similar header type. You don't need to refer to the fields of this record.

```
PickListPtr = ^PickList;
PickList =
  object(CommandWindow)
    ...
  end;
```

The PickList object.

Variables

```
PickCommands : CommandProcessor;
```

The default CommandProcessor for a PickList. This variable is initialized by OPPICK's initialization block. See OPPICK.ICD for details of the key assignments.

AddMoreHeader

Declaration

```
procedure AddMoreHeader(PromptStr : String80;
                        Posn : HeaderPosType;
                        UpStr, DnStr, SepStr : String10;
                        UpPos, DnPos, SepPos : Byte);
```

Purpose

Add a specialized more header to indicate more pick items.

Description

This header is useful when the pick list has more items than will fit into the window at once and scroll bars are not desired. It puts a message like "Up/Down for More" into the PickList's window frame. The actual message is dynamically modified depending on the highlight bar's position within the list.

AddMoreHeader makes a call to wFrame.AddHeader after it initializes internal data structures to keep track of the more header. PickList then manages the modification and display of this header.

A PickList may have only one of these headers at a time. If more than one is added, PickList will dispose of the previous one before adding the new one.

Posn specifies the frame-relative position of the header (heTL, heTC, heTR, heBL, heBC, or heBR).

The remaining strings and byte values specify the components of the header, which is comprised of four parts. It's easier to describe these by using an example. Suppose we want the header to read:

```
'Up/Down for More'
```

PromptStr defines the fixed text and place-holding positions for the varying portions. In this example it could be:

```
'xxxxxxx for More'
```

The xxxxxxx positions contain arbitrary characters since they are just place-holders. However, it's a good idea to use the original string for PromptStr.

UpStr--'Up' in this example--defines the string inserted into PromptStr when there are more items above the current window. UpPos specifies the position where the string is inserted into PromptStr--1 in this case.

DnStr and DnPos are defined similarly and would have the values 'Down' and 4 in this case.

SepStr is displayed only when both UpStr and DnStr are shown. In this case it would equal '/' and SepPos would be 3.

When the list's scroll position is such that one of these strings isn't appropriate, its position is filled with spaces. If none of the dynamic strings are needed (because the list completely shows in the pick window), the header is not drawn at all.

AddMoreHeader checks for various errors, including bad string positions and insufficient memory to allocate string buffers. These errors can be detected by calling GetLastError.

Example

```
PizzaTop.AddMoreHeader(' .. More', heBR, #24, #25, '', 2, 3, 0);
```

Adds a bottom-right more header to the pick window. The two dots in the header string will be overwritten by the UpStr and DnStr characters (up and down arrows in this case). The up arrow will be written to the second character position of the header, the down arrow will be written to the third character position. There is no separator string for this header.

See Also

Frame.AddHeader

Declaration
```
procedure AddSearchHeader(MaxLen : Byte; Posn : HeaderPosType);
```

Purpose
Add a specialized search string header.

Description
This header is appropriate only if you intend to enable the PickStringSearch or PickAltStringSearch search modes. The header will display the current value of the incremental search string.

AddSearchHeader makes a call to wFrame.AddHeader after it initializes internal data structures. PickList then manages the modification and display of this header.

MaxLen specifies the maximum length to display. There's no point in making MaxLen longer than pkMaxSearchLen, which specifies the longest search string OPPICK will store.

Posn specifies the frame-relative position of the header (heTL, heTC, heTR, heBL, heBC, or heBR).

You can have at most one search string header per pick list.

If there is insufficient memory, AddSearchHeader will generate the error epFatal+ecOutOfMemory.

Example
```
PizzaTop.SetSearchMode(PickStringSearch);
PizzaTop.AddSearchHeader(20, heTL);
```

Sets the string search mode and turns on a search header. As alpha characters are entered, a search string is displayed at the top-left header postition.

See Also
SetSearchMode

Declaration
```
procedure AllocateSelectSet(MaxItems : Word);
```

Purpose
Allocate select set for the specified number of items.

Description
This method is intended for multiple choice lists. PickList will call it automatically if you don't. There are two reasons why you might want to call it explicitly. If the number of the items on the list might grow you can allocate more than you need initially to avoid heap fragmentation when the list grows later. The directory list object (derived from PickList) uses this technique, since it can't predict the exact number of files it will need to manage.

Another time to call AllocateSelectSet is when you want to mark certain items selected prior to calling the Process method. See the introductory section "Multiple Choice Lists" for more information.

AllocateSelectSet allocates about (MaxItems+8) div 8 bytes of heap space. All of the bits are initialized to the "clear" state. If you call AllocateSelectSet a second time with fewer items than are already allocated, AllocateSelectSet does nothing. If you call it with more items, it deallocates the old BitSet and allocates a new one. In this case the settings in the old BitSet will be lost.

Example
```
PizzaTop.AllocateSelectSet(5);
```

Allocate a select set capable of managing 5 items.

See Also
Process

ChangeCommandHandler / ChangeItemWidth

Declaration

```
procedure ChangeCommandHandler(CommandHandler : pkGenlProc);
```

Purpose

Change between single and multiple choice.

Description

OPPICK provides two procedures that can be passed as parameters to this routine: SingleChoice and MultipleChoice. This method initializes internal data fields that indicate the routines Process should use the next time it is called.

Example

```
PizzaTop.ChangeCommandHandler(MultipleChoice);
PizzaTop.SetSelectMarker('<', '>');
```

Changes PizzaTop to use the MultipleChoice command handler.

See Also

Init

Declaration

```
procedure ChangeItemWidth(ItemWidth : Byte);
```

Purpose

Change the item width.

Description

Call this routine when changing the format of the pick list. The DirList object used for choosing from DOS file directories provides a number of possible formats–filename only, filename plus size, filename plus size plus date and time, etc.–and that object must call ChangeItemWidth whenever the format is changed.

ChangeItemWidth is generally accompanied by a call to ChangeStringProc and is followed by calls to OptimizeSize and Draw.

Example

```
PizzaTop.ChangeItemWidth(10);
PizzaTop.OptimizeSize;
PizzaTop.Draw;
```

Changes PizzaTop to use an item width of 10 characters; the window will be collapsed down to the new size of the list.

See Also

ChangeStringProc OptimizeSize

Declaration
```
procedure ChangeNumItems(NumItems : Word);
```

Purpose
Change the number of items to display.

Description
The next time the pick window is drawn after this call, it will be updated to use the new number of items. Changing the number of items doesn't automatically change the size of the pick window, but it may change the arrangement of items within the window.

Examples
```
{$F+}
procedure PizzaTopping(Item : Word; Mode : pkMode;
                       var IType : pkItemType; var IString : String;
                       PickPtr : PickListPtr);
  {-Sets item string and item options (if any)}
begin
  case Item of
    1 : IString := ' Pepperoni';
    2 : IString := ' Sausage';
    3 : IString := ' Mushrooms';
    4 : IString := ' Anchovies';
    5 : IString := ' Onions';
  end;
end;
{$F-}
...
PizzaTop.ChangeNumItems(3);
```
Reduces the number of items in PizzaTop to three. Items four and five are no longer displayed with the pick list.
```
PizzaTop.ChangeNumItems(6);
```
Incorrect! Our procedure only goes up to five. Garbage or an unpredictable value will be displayed in the sixth position. It's permissible to increase the size of the list only when the string procedure can handle that increase.

See Also
OptimizeSize

Declaration
```
procedure ChangeOrientation(Orientation : pkGenlProc);
```

Purpose
Change the orientation.

Description
Acceptable values for Orientation include PickVertical, PickHorizontal, or PickSnaking. ChangeOrientation calls the routine specified as its parameter and this routine changes a number of internal procedure pointers to create the behavior associated with the new orientation. The display will be updated the next time Draw or Process is called.

Example
```
PizzaTop.ChangeOrientation(PickHorizontal);
```
Changes PizzaTop's orientation to PickHorizontal.

See Also
GetOrientation

ChangeStringProc / ClearSelected

Declaration

```
procedure ChangeStringProc(StringProc : pkStringProc);
```

Purpose

Change the string procedure.

Description

This procedure does something only if the ItemString procedure has not been overridden. The default ItemString calls the routine specified by the StringProc procedure pointer.

StringProc is a user-defined routine that must be compiled under the FAR model using the {$F+} directive, must be a global procedure, and must match the following prototype declaration:

```
{$F+}
procedure MyStringProc(Item : Word;
                       Mode : pkMode;
                       var IType : pkItemType;
                       var IString : String;
                       PickPtr : PickListPtr);
```

For further information see the introduction to this section.

Example

```
PizzaTop.ChangeStringProc(OtherToppings);
```

Changes PizzaTop's StringProc to a procedure named OtherToppings.

See Also

CompleteItemString ItemString

Declaration

```
procedure ClearSelected;
```

Purpose

Clear selected items.

Description

Meaningful only for multiple choice pick lists. Marks all items unselected. The screen is not updated until Draw or Process is called.

The select set must be allocated before calling this method or it does nothing. Note that the select set is cleared automatically when it is first allocated.

Example

```
PizzaTop.ClearSelected;
```

Assuming that we're processing multiple selections, this will clear all current selections.

See Also

AllocateSelectSet MarkAllSelected

Declaration

```
procedure CompleteItemString(Item : Word; Mode : pkMode;
                             var IType : pkItemType;
                             var IString : String); virtual;
```

Purpose

Supplies, marks, and pads item string.

Description

OPPICK calls this virtual method to generate each item string in order to display it or compare against it during a search. CompleteItemString in turn calls the virtual method ItemString, which calls the user string procedure. For multiple choice lists, CompleteItemString concatenates left and right markers (specified with SetSelectMarker) if the item is selected, or pads it with window background characters if not. CompleteItemString then pads the string with background characters as specified by the SetPadSize method.

For further discussion of the parameters passed to CompleteItemString, see the discussion of "The User-Defined String Procedure" at the beginning of the OPPICK section.

See Also

ItemString
SetPadSize
SetSelectMarker

Declaration

```
procedure DeselectItem(Item : Word); virtual;
```

Purpose

Deselect an item.

Description

Useful only for multiple choice lists. This method clears the select set's bit for the item. The pick list display will be updated the next time it is drawn.

This method is virtual so that derived objects (DirList for example) can ignore attempts to deselect or select certain items.

Example

```
repeat
  PizzaTop.Process;
  case PizzaTop.GetLastCommand of
    ccUser0 :
      for I := 1 to PizzaTop.GetNumItems do
        if PizzaTop.ItemIsSelected(I) then
          PizzaTop.DeselectItem(I)
        else
          PizzaTop.SelectItem(I);
    {other exit commands}
  end;
until Finished;
```

Whenever the key associated with ccUser0 is pressed, all selected items are deselected and all selected items are deselected.

See Also

ClearSelected
SelectItem

Done / GetDefaultChoice

Declaration
```
destructor Done; virtual;
```

Purpose
Dispose of PickList.

Description
If a select set has been allocated for a multiple choice list, it is deallocated. If a more header has been added, its internal string storage is deallocated. Then the PickList's parent destructor, CommandWindow.Done, is called.

Example
```
PizzaTop.Done;
```
Deallocates all heap space used by PizzaTop.

See Also
RawWindow.Done

Declaration
```
function GetDefaultChoice : Word;
```

Purpose
Return default choice number.

Description
If the pkSetDefault option is enabled, GetDefaultChoice returns the item number that was current at the beginning of the most recent call to Process. Otherwise it returns the value specified by the last call to SetDefaultChoice, or 1 by default.

This method is useful primarily within the user string procedure, where you can highlight the default choice so that the user can easily return to it. You might also want to use GetDefaultChoice in combination with SetInitialChoice to restore the current item to the previous default after the user exits Process with a ccQuit command.

Note that GetDefaultChoice and GetLastChoice often return different values. GetLastChoice always returns the last item that had the selection bar over it.

Example
```
procedure PizzaList.ItemString(Item : Word; Mode : pkMode;
                               var IType : pkItemType;
var
  IString : String);
begin
  case Item of
    1 : IString := ' Pepperoni';
    2 : IString := ' Sausage';
    3 : IString := ' Mushrooms';
    4 : IString := ' Anchovies';
    5 : IString := ' Onions';
  end;
  if Item = GetDefaultChoice then
    IType := pkAlternate;
end;
```
Forces the default item to display in the alternate video attribute.

See Also
GetLastChoice
SetDefaultChoice
SetInitialChoice

Declaration
```
function GetItemCols : Byte;
```

Purpose
Return number of visible item columns.

Description
The number of item columns is calculated from the total number of items, the width of each item string, and the size of the pick window. This function returns the result of that calculation.

Example
```
WriteLn('This window will hold ', PizzaTop.GetItemCols, ' columns');
```
Displays number of item columns in pick list.

See Also
Init

Declaration
```
function GetItemString(Item : Word) : string;
```

Purpose
Return specified item string in search format.

Description
GetItemString calls the CompleteItemString virtual method to get the name of the specified Item. When it does so, it specifies pkSearch mode, which prevents the default CompleteItemString method from adding selection markers or padding the string.

A properly designed user string procedure will not embed flexwrite control characters in the returned string when pkSearch mode is specified.

Example
```
WriteLn('The name of the first item is ', PizzaTop.GetItemString(1));
```
Writes the name of the first pick item.

See Also
CompleteItemString GetLastChoiceString

GetLastChoice / GetLastChoiceString / GetNumItems

Declaration

```
function GetLastChoice : Word;
```

Purpose

Return last selected item number.

Description

The method always returns the item number that most recently has had the selection bar over it. Its result is valid at any time it can be called–within a user string procedure, a move procedure, a string search function, and after a Process call returns.

Note that GetLastChoice returns the most recently highlighted item even if the user presses <Esc> to exit a Process call. It is therefore important to check the value of GetLastCommand before counting on GetLastChoice.

Example

See the example program at the beginning of this section.

See Also

CommandWindow.GetLastCommand GetDefaultChoice

Declaration

```
function GetLastChoiceString : string;
```

Purpose

Return last choice in search format.

Description

GetLastChoiceString calls GetItemString with a parameter of GetLastChoice.

Example

```
PizzaTop.Process;
if PizzaTop.GetLastCommand = ccSelect then
  WriteLn('You chose item ', PizzaTop.GetLastChoice,
          ' - ', PizzaTop.GetLastChoiceString);
```

Displays both the number and string of the selected item.

See Also

GetItemString

Declaration

```
function GetNumItems : Word;
```

Purpose

Return number of pick items.

Description

The number of items is specified by the PickList's initial constructor call or by a subsequent call to ChangeNumItems.

Example

```
WriteLn('This pick list has ', PizzaTop.GetNumItems, ' items');
```

Displays number of items in the pick list.

See Also

ChangeNumItems Init

Declaration
```
function GetOrientation : Byte;
```

Purpose
Return the orientation code.

Description
The PickList's orientation is specified by the initial constructor call or by a subsequent call to ChangeOrientation. GetOrientation returns one of the values pkNoOrient, pkVertical, pkHorizontal, or pkSnaking. pkNoOrient is used internally and indicates that an invalid orientation routine was specified. You will never see this value if proper error checking is in place.

Example
```
case PizzaTop.GetOrientation of
  pkVertical   : {do something};
  pkHorizontal : {do something};
  pkSnaking    : {do something};
end;
```
Takes some action based on the current pick orientation.

See Also
ChangeOrientation Init

Declaration
```
function GetSelectedCount : Word;
```

Purpose
Return number of items selected.

Description
For single choice pick lists, this always returns zero. For multiple choice lists, it returns the number of items currently selected.

Example
```
PizzaTop.Process;
if PizzaTop.GetLastCommand = ccSelect then
  if PizzaTop.GetSelectedCount > 0 then begin
    WriteLn('You selected:');
    PizzaTop.InitSequence(SelectedItem);
    while PizzaTop.HaveSelected(SelectedItem) do begin
      WriteLn(PizzaTop.GetItemString(SelectedItem));
      PizzaTop.NextSelected(SelectedItem);
    end;
  end else
    Writeln('No selections');
```
After a Process call, verifies that some items were selected before performing an action on all such items.

See Also
AllocateSelectSet ClearSelected

HaveSelected / Init

Declaration

```
function HaveSelected(Item : Word) : Boolean;
```

Purpose

Return True if there is a selected item.

Description

This method is used in combination with InitSequence and NextSelected to scan through all selected items in a multiple choice list. HaveSelected returns True whenever Item is less than or equal to the number of items (GetNumItems), so it isn't much use by itself.

Example

```
var
  N : Word;
...
PizzaTop.InitSequence(N);
while PizzaTop.HaveSelected(N) do begin
  WriteLn('Selected item number ', N);
  PizzaTop.NextSelected(N);
end;
```

Shows the proper sequence for retrieving selected items from a pick list.

See Also

InitSequence NextSelected

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte;
                 ItemWidth : Byte;
                 NumItems : Word;
                 StringProc : pkStringProc;
                 Orientation : pkGenlProc;
                 CommandHandler : pkGenlProc);
```

Purpose

Initialize a pick window.

Description

This constructor instantiates a PickList using the default window options (DefWindowOptions), the default color set (DefaultColorSet), and the default pick options (DefPickOptions). See InitDeluxe for more details.

Example

```
if not PizzaTop.Init(35, 5, 45, 10,     {Initial window coordinates}
                     20,                {Column width per item}
                     NumPizzaToppings,  {Number of picklist items}
                     PizzaTopping,      {User string procedure}
                     PickVertical,      {Orientation procedure}
                     SingleChoice)      {Command handler}
then begin
  WriteLn('Failed to Init PickList,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates PizzaTop using default window colors and options. The pick list will have vertical orientation, use PizzaTopping as the string function, and return a single choice.

See Also

InitAbstract InitDeluxe
InitCustom

Declaration

```
constructor InitAbstract(X1, Y1, X2, Y2 : Byte;
                         var Colors : ColorSet;
                         Options : LongInt;
                         ItemWidth : Byte;
                         NumItems : Word;
                         Orientation : pkGenlProc;
                         CommandHandler : pkGenlProc);
```

Purpose

Constructor to be called by derived types that override the ItemString method.

Description

This constructor works like InitCustom except that no user string procedure is supplied. The caller must have overridden PickList's ItemString and/or CompleteItemString methods so that ItemString no longer generates a call to the user-defined string procedure. Otherwise a runtime error (error code 211) will occur during the first call to ItemString.

Example

See the discussion of "The User-Defined String Procedure" near the beginning of this section for a complete example.

See Also

InitDeluxe

Declaration

```
constructor InitAbstractDeluxe(X1, Y1, X2, Y2 : Byte;
                               var Colors : ColorSet;
                               Options : LongInt;
                               ItemWidth : Byte;
                               NumItems : Word;
                               Orientation : pkGenlProc;
                               CommandHandler : pkGenlProc;
                               PickOptions : Word);
```

Purpose

Constructor to be called by derived types that override the ItemString method, with custom pick options.

Description

This constructor works like InitAbstract, except that custom pick options (ones that differ from DefPickOptions) can be specified at the time of the call.

Example

```
if not PickList.InitAbstractDeluxe(X1, Y1, X2, Y2,
                                   DefaultColorSet,
                                   PickWindowOptions,
                                   11,
                                   NumPizzaToppings,
                                   PickVertical,
                                   SingleChoice,
                                   pkStick+pkMousePage+pkSetDefault)
then
  Fail;
```

The constructor of a derived type calls its parent's constructor with the specific pick list options pkStick + pkMousePage + pkSetDefault.

See Also

InitDeluxe

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet;
                       Options : LongInt;
                       ItemWidth : Byte;
                       NumItems : Word;
                       StringProc : pkStringProc;
                       Orientation : pkGenlProc;
                       CommandHandler : pkGenlProc);
```

Purpose

Initialize a pick window with custom window options.

Description

This constructor instantiates a PickList using the specified window Options, the specified Colors, and default pick options (DefPickOptions). See InitDeluxe for more details.

Example

```
PickWindowOptions := DefWindowOptions or wBordered;
if not PizzaTop.InitCustom(35, 5, 45, 10,     {Initial window coordinates}
                           DefaultColorSet,   {ColorSet to use}
                           PickWindowOptions, {Window options}
                           20,                {Columns per item}
                           NumPizzaToppings,  {Number of picklist items}
                           PizzaTopping,      {Item string procedure}
                           PickVertical,      {Orientation procedure}
                           SingleChoice)      {Command handler}
then begin
  WriteLn('Failed to Init PickList,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates PizzaTop with default window colors but using custom window options to create a bordered window.

See Also

RawWindow.InitCustom

Declaration

```
constructor InitDeluxe(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet;
                       Options : LongInt;
                       ItemWidth : Byte;
                       NumItems : Word;
                       StringProc : pkStringProc;
                       Orientation : pkGenlProc;
                       CommandHandler : pkGenlProc;
                       PickOptions : Word);
```

Purpose

Initialize a pick window with custom window and pick options.

Description

This is the most general constructor for a pick window. All of the other PickList constructors make specific calls to this one.

X1, Y1, X2, and Y2 specify the active window coordinates within which the pick list will appear.

InitDeluxe plucks certain fields out of the specified ColorSet to use while drawing the pick list. Besides those used by windows in general, InitDeluxe stores the following fields within the PickList instance:

```
TextColor, TextMono             - pkNormal items, when not selected
SelItemColor, SelItemMono       - pkNormal items, when selected
AltItemColor, AltItemMono       - pkAlternate items, when not selected
AltSelItemColor, AltSelItemMono - pkAlternate items, when selected
ProItemColor, ProItemMono       - pkProtected items (never selected)
```

Note that "selected" means the item that currently has the selection bar over it, not a marked item in a multiple choice pick list. Also note that by default, items are displayed in uniform colors–flexwriting is not activated.

Options are the bit-mapped flags, such as wBordered, that apply to all RawWindow-derived objects. See the RawWindow Constants section for more details. InitDeluxe forces the wUserContents flag on, since PickList provides an UpdateContents method that can regenerate the window's contents on request.

ItemWidth is the number of screen columns that each item will use. Be sure to reserve space here for the item string itself, for padding specified by SetPadSize, and for multiple choice selection markers specified by SetSelectMarker. InitDeluxe determines how many columns of items to display in the pick window by dividing the window width by ItemWidth. When each item is displayed, it is clipped or padded to ItemWidth.

NumItems is the total number of items that the PickList will manage. The user-defined string function must be capable of returning an item name for each item number in the range from 1 to NumItems.

StringProc supplies the name of a user-defined string procedure. This procedure must be compiled with the FAR model, must be global, and must have parameters matching the following prototype:

```
procedure MyStringProc(Item : Word;
                       Mode : pkMode;
                       var IType : pkItemType;
                       var IString : String;
                       PickPtr : PickListPtr);
```

See the discussion of "The User-Defined String Procedure" near the start of this section for more information.

Orientation specifies the list's orientation. It can take one of the values PickVertical, PickHorizontal, or PickSnaking. See "Pick List Orientations" in this section's introduction for details.

CommandHandler specifies a single choice list (SingleChoice) or multiple choice (MultipleChoice). See "Multiple Choice Lists" for further information. Be careful not to reverse the Orientation and CommandHandler parameters: the compiler will let you do so, but you'll get a constructor error at runtime (epFatal+ecBadPickOrient).

PickOptions provides options specific to a pick list. See the Constants declarations for details.

The constructor calculates the pick list configuration–number of columns, items in each row, etc.–by evaluating the item width, the window width, the number of items, and the list orientation. If the pkMinHeight option is enabled, InitDeluxe will reduce the window height if it exceeds the number of rows of items.

The PickCommands command processor is assigned to the PickList. The list is assigned a unit code of ucPick (for the purposes of identifying it to error and help handlers). The hardware cursor is set to a default mode of cuHidden.

InitDeluxe allocates no heap space beyond that required by its parent, CommandWindow.InitCustom. Note especially that InitDeluxe does not allocate the BitSet required for managing multiple choice pick lists. This BitSet is not allocated until AllocateSelectSet, Process, or UpdateContents is called.

If the constructor fails, check the InitStatus variable for a detailed error code.

Example

```
var
  PickWindowOptions : LongInt;
  PickListOptions : Word;
...
PickWindowOptions := DefWindowOptions or wBordered;
PickListOptions := DefPickOptions and not pkMinHeight;
if not PizzaTop.InitDeluxe(35, 5, 45, 15,     {Initial window coordinates}
                           DefaultColorSet,   {ColorSet to use}
                           PickWindowOptions, {Window options}
                           20,                {Columns per item}
                           NumPizzaToppings,  {Number of picklist items}
                           PizzaTopping,      {Item string procedure}
                           PickVertical,      {Orientation procedure}
                           SingleChoice,      {Command handler}
                           PickListOptions)   {List options}
then begin
  WriteLn('Failed to Init PickList,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates PizzaTop with list options that prevent the constructor from changing the window size. Given the 5 pizza toppings in our example and the window height of 11 (5..15), there will be blank rows below the list.

See Also

ChangeCommandHandler
ChangeItemWidth
ChangeNumItems
ChangeOrientation
ChangeStringProc
CommandWindow.Init
RawWindow.Init
SetPadSize
SetPickFlex
SetSelectMarker

Declaration
```
procedure InitSequence(var Item : Word);
```

Purpose
Prepare to get first selected item.

Description
This method is used in combination with HaveSelected and NextSelected to scan through all selected items in a multiple choice list. InitSequence initializes Item to the index of the first selected value. If there are no selected items, it sets Item to one more than the number of items in the list.

Example
```
var
  N : Word;
...
PizzaTop.InitSequence(N);
while PizzaTop.HaveSelected(N) do begin
  WriteLn('Selected item number ', N);
  PizzaTop.NextSelected(N);
end;
```

Shows the proper sequence for retrieving selected items from a pick list.

See the introductory section "Multiple Choice Lists" for a complete example program.

See Also
HaveSelected NextSelected

Declaration
```
function IsProtected(Item : Word) : Boolean;
```

Purpose
Return True if Item is protected from selection.

Description
IsProtected calls CompleteItemString in pkGetType mode (which informs the string procedure that only the type of the item is needed). If the string procedure returns the pkProtected type, IsProtected returns True. The list's selection bar is never allowed to land on a protected item.

This method is primarily used by PickList's own methods. Presumably your application already knows which items are protected since it is your string procedure that defines them as such. However, if users of the application can somehow protect items, you may need to call this method yourself.

See Also
NumSelectableItems

ItemIsSelected / ItemSearch / ItemString

Declaration

```
function ItemIsSelected(Item : Word) : Boolean;
```

Purpose

Return True if item is selected.

Description

This method is useful only for multiple choice lists.

Example

```
repeat
  PizzaTop.Process;
  case PizzaTop.GetLastCommand of
    ccUser0 :
      for I := 1 to PizzaTop.GetNumItems do
        if PizzaTop.ItemIsSelected(I) then PizzaTop.DeselectItem(I)
        else PizzaTop.SelectItem(I);
    {...other exit commands...}
  end;
until Finished;
```

Whenever the key associated with ccUser0 is pressed, all selected items are deselected and all selected items are deselected.

See Also

DeselectItem SelectItem

Declaration

```
function ItemSearch : Boolean; virtual;
```

Purpose

Called to search list each time an alphanumeric character is entered.

Description

PickList's Process method calls this virtual method to search the list for an alphanumeric match. ItemSearch in turn calls the search function specified via the SetSearchMode method.

See the discussion of "String Searching" in the introduction to this section for more details.

See Also

SetSearchMode

Declaration

```
procedure ItemString(Item : Word; Mode : pkMode; var IType : pkItemType;
                     var IString : String); virtual;
```

Purpose

Supplies each item string when the list is displayed or searched.

Description

This virtual method may be overridden by derived types to implement a specific kind of list. PickList's implementation of ItemString simply passes the call on to another user-defined string procedure.

See the discussion of "The User-Defined String Procedure" near the start of this section for more information and for complete examples.

See Also

ChangeStringProc InitDeluxe
CompleteItemString

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a pick list from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to PickList.Store. Load first calls CommandWindow.Load to read the underlying CommandWindow. Then it reads data specific to the PickList.

The stream registration procedure for a PickList is PickListStream. There are also a number of procedure pointers that must be registered independently. The example program below demonstrates how. See Store for more details.

Example

```
{... same as introductory example ...}
var
  S : BufIdStream;
begin
  {Load PizzaTop from a stream}
  S.Init('PIZZA.STM', SOpen, 1024);
  S.RegisterHier(PickListStream);
  S.RegisterPointer(ptPickVertical, @PickVertical);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.RegisterPointer(ptPickCharSearch, @PickCharSearch);
  S.RegisterPointer(1000, @PizzaTopping);
  S.Get(PizzaTop);
  S.Done;

  {Pick an item}
  PizzaTop.Process;
  PizzaTop.Erase;
  if PizzaTop.GetLastCommand = ccSelect then
    WriteLn('You chose item ', PizzaTop.GetLastChoice,
            ' - ', PizzaTop.GetLastChoiceString)
  else
    Writeln('No choice was made');
  PizzaTop.Done;
end.
```

Instantiates PizzaTop from the stream created by the Store example. See Store for more information about type registration.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

CommandWindow.Load Store

MarkAllSelected / NextSelected

Declaration

```
procedure MarkAllSelected;
```

Purpose

Mark all items selected.

Description

Applies only to multiple choice lists. Be careful to assure that the select BitSet is allocated before calling MarkAllSelected. See AllocateSelectSet for more information.

Example

```
PizzaTop.AllocateSelectSet(5);
PizzaTop.SetSelectMarker('<', '>');
PizzaTop.MarkAllSelected;
PizzaTop.Process;
```

PizzaTop starts off with all items selected.

See Also

AllocateSelectSet
ClearSelected
SelectItem

Declaration

```
procedure NextSelected(var Item : Word);
```

Purpose

Position Item at the next selected item beyond the initial value.

Description

This method is used in combination with InitSequence and HaveSelected to scan through all selected items in a multiple choice list. NextSelected scans the select set starting one beyond Item to find the next selected item. If there are no more selected items, it sets Item to one more than the number of items in the list.

Example

```
var
  N : Word;
...
PizzaTop.InitSequence(N);
while PizzaTop.HaveSelected(N) do begin
  WriteLn('Selected item number ', N);
  PizzaTop.NextSelected(N);
end;
```

Shows the proper sequence for retrieving selected items from a pick list.

See the introductory section "Multiple Choice Lists" for a complete example program.

See Also

HaveSelected
InitSequence

Declaration

```
function NumSelectableItems : Word;
```

Purpose

Return number of items that are not protected.

Description

This function calls IsProtected for every item on the list and counts the number that return False. For lists without protected items, NumSelectableItems will equal GetNumItems.

Most programs know in advance how many items are protected and how many are selectable. Some applications may give the user or some other outside factor control over the protected items, however. If this is the case, it's important to assure that NumSelectableItems returns a non-zero value before calling the PickList's Process method.

See "Protected Items" in this section's introduction for more information.

Example

```
if PizzaTop.NumSelectableItems > 0 then begin
  PizzaTop.Process;
  ...
end else
  WriteLn('All items are protected');
```

See Also

IsProtected Process

OptimizeSize

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure OptimizeSize;
```

Purpose

Resize window to surround items.

Description

OptimizeSize adjusts the pick window's width and height to best "fit" the list. It takes into account the current number of items, the item width, the originally requested window size, and the window adjustment limits. It applies a heuristic algorithm to arrive at a new size. Note that the window's upper left corner always remains fixed; only the width and height change.

It makes sense to call this routine only after changing the list's number of items or item width.

First, the width is increased to hold as many whole columns of items as possible without exceeding a) the original width specified when the PickList was constructed, b) the window's active size limit, c) the frame's maximum x-coordinate, or d) the frame's maximum size limit. If possible, the width is set to a value which exactly fits an integral number of columns.

Second, the height is adjusted to match the number of item rows as closely as possible. Again, the adjustment is constrained by the window's active size limits, the frame's coordinate limits, and the frame's size limits.

The optimization takes into account the window's shadows only if the pkOptFullExtent flag is set. By enabling this option you can keep OptimizeSize from pushing the shadows outside of the frame's clipping rectangle.

Finally, OptimizeSize calls StackWindow.ResizeWindow to adjust the window. If the window is current, the changes will show up immediately. If the window is not active, the change will appear the next time the window is drawn. Remember that temporary heap buffers may be required for a virtual screen and that heap fragmentation may result from resizing the window.

OPPICK does not change the current item number thoughout the optimization process.

The window must be flagged as resizeable (wResizeable) and must not be zoomed. If these prerequisites aren't met, OptimizeSize exits immediately. (It doesn't generate an error.) The window must either be inactive or, if active, the current window. Otherwise, ResizeWindow generates an error.

Example

```
PizzaTop.InitCustom(35, 5, 45, 10, DefaultColorSet,
                    DefWindowOptions or wBordered,
                    11, NumPizzaToppings, PizzaTopping,
                    PickVertical, SingleChoice);
...
ChangeNumItems(10);
OptimizeSize;
Process;
...
```

After increasing the number of items from NumPizzaToppings (5) to 10, we call OptimizeSize, which will increase the height of the window to make all items visible at once. If we had originally made the window wider than required (by changing the X2 parameter from 45 to 50, for example), OptimizeSize would reduce this coordinate so that the window exactly fit the items.

See Also

ChangeItemWidth
ChangeNumItems
StackWindow.ResizeWindow

Declaration
```
function pkOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This function tests the pick list options, which are described in the Constant declarations near the beginning of this section.

Example
```
if PizzaTop.pkOptionsAreOn(pkUseItemForTopic) then
  HelpWindow.SetTopic(HelpIndex+100)
else
  Exit;
```
Within a user-written help routine, tests whether the pick list is passing an item number as the help topic index. If so, it adds 100 to that index and assigns it as the current topic in the help window.

Declaration
```
procedure pkOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadPickOptions (see the Constants section, above).

Example
```
PizzaTop.pkOptionsOff(pkMinHeight+pkSetDefault);
```
Turns off height minimization and automatic storage of the default item.

Declaration
```
procedure pkOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadPickOptions (see the Constants section, above).

Example
```
PizzaTop.pkOptionsOn(pkMinHeight+pkSetDefault);
```
Turns on height minimization and automatic storage of the default item.

PreMove / Process

Declaration

```
procedure PreMove; virtual;
```

Purpose

Called just prior to getting each keyboard command.

Description

This virtual method may be overridden by derived types to implement actions to be executed before each command. PickList's implementation of PreMove simply passes the call on to a user-defined routine. You can specify a particular routine by using the SetMoveProc method.

See the discussion of "The User Move Hook" near the start of this section for more information and complete examples.

See Also

SetMoveProc

Declaration

```
procedure Process; virtual;
```

Purpose

Process pick commands leading to a choice.

Description

The PickList Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Pick List Commands" at the start of this section.

If there are no pick items (GetNumItems returns zero), Process clears the pick window and exits with the warning epWarning + ecNoPickItems.

If the list is set to process multiple choices and the selection BitSet has not yet been allocated, Process allocates a set large enough to hold the current number of pick items. If there is insufficient heap space, Process exits with the error epFatal + ecInsufficientMemory.

Process recomputes the list configuration (number of columns, items in each column, etc.) in case the window has been resized or some other list parameter has been changed since the list was initialized.

If the pkSetDefault option is enabled, Process saves the currently selected item in an instance data field, where it can be accessed by the GetDefaultChoice method.

It is a programming error to call Process when all list items are protected, but Process will still recognize the exit commands in this case.

Process calls at least one, and perhaps several, user-defined routines. The string procedure, the premove procedure, and the string search function are specific to the PickList. The error handling procedure is inherited from the CommandWindow. And the context-sensitive help and keyboard background processing procedures are inherited from the CommandProcessor.

Example

For a complete example, see the introduction to this section.

See Also

CommandWindow.Process GetLastChoice

Declaration
```
procedure SelectItem(Item : Word); virtual;
```

Purpose
Select an item.

Description
Useful only for multiple choice lists. This method selects the specified item. The pick list display will be updated the next time it is drawn.

This method is virtual so that derived objects (DirList for example) can ignore attempts to select certain items.

Example
```
repeat
  PizzaTop.Process;
  case PizzaTop.GetLastCommand of
    ccUser0 :
      for I := 1 to PizzaTop.GetNumItems do
        if PizzaTop.ItemIsSelected(I) then
          PizzaTop.DeselectItem(I)
        else
          PizzaTop.SelectItem(I);
    {other exit commands}
  end;
until Finished;
```

Whenever the key associated with ccUser0 is pressed, all selected items are deselected and all selected items are deselected.

See Also
DeselectItem MarkAllSelected

Declaration
```
procedure SetChoice(Choice, FirstChoice : Word);
```

Purpose
Set initial choice with scrolling control.

Description
Once the configuration of a particular list is defined, PickList uses two parameters to decide how to draw the pick window. One of these, Choice, is the index of the item to be highlighted as current. The other, FirstChoice, specifies the index of the item that appears in the upper left corner of the window. As the user moves around the window with cursor keys, Choice varies but FirstChoice remains fixed until the list must scroll.

When a PickList is initialized, both Choice and FirstChoice are set to 1. SetChoice gives you a way to set the initially selected item and to control the window's scrolling at the same time.

If Choice and FirstChoice are separated by more items than will fit into one window, the PickList will adjust the value of FirstChoice to allow Choice to appear within the window.

If Choice is zero or greater than the number of items, or if the item referred to by Choice is protected, SetChoice exits immediately.

Example
```
PizzaTop.SetChoice(4, 3);
PizzaTop.Process;
```

Sets item 4 as the initial choice and selects item 3 as the first item visible in this window.

See Also
SetInitialChoice

SetDefaultChoice / SetInitialChoice

Declaration
```
procedure SetDefaultChoice(Choice : Word);
```

Purpose
Set the default choice number.

Description
This method simply stores the specified item number, Choice, in an internal instance variable where it can be referenced by the GetDefaultChoice method. This is useful if the list's string procedure is set up to highlight a particular item as the default or preferred choice.

If the pkSetDefault option is enabled, the Process method will overwrite the same internal variable each time Process is called.

Do not confuse this method with SetChoice or SetInitialChoice, which actually set the current item.

Example
```
PizzaTop.pkOptionsOff(pkSetDefault);
PizzaTop.SetDefaultChoice(5);
```

Assures that Process won't fiddle with the default choice and then sets the default to item number 5. If the PizzaTopping string procedure ends with the following line:

```
if PickPtr^.GetDefaultChoice = Item then
  IType := pkAlternate;
```

then Process will draw the fifth item in the alternate video attribute.

See Also
GetDefaultChoice

Declaration
```
procedure SetInitialChoice(Choice : Word);
```

Purpose
Set initial choice without scrolling if possible.

Description
Works like SetChoice, except that it doesn't change the value of FirstChoice. Whenever possible, the new Choice will be highlighted without scrolling the pick window. The new Choice will become visible the next time Draw or Process is called.

Example
```
PizzaTop.SetInitialChoice(4);
PizzaTop.Process;
```

Sets an initial choice just before calling Process. When the pick list is drawn the fourth choice will already be selected.

See Also
SetChoice

Declaration

```
procedure SetMoveProc(MP : pkGenlProc);
```

Purpose

Set user move procedure.

Description

Stores the address of the specified procedure in an internal data field. The move procedure will be used the next time Process is called.

Note that OPPICK interfaces a properly declared procedure, NoMoveAction, which you can pass to SetMoveProc in order to disable a previously activated move procedure.

See the discussion of "The User Move Hook" in the introduction to this section for more information.

Example

```
PizzaTop.SetMoveProc(Suggestions);
```

Stores the address of the Suggestions procedure for reference by Process.

See Also

PreMove

Declaration

```
procedure SetPadSize(LeftPad, RightPad : Byte);
```

Purpose

Set number of characters for padding of pick items.

Description

After an item string is returned by the user-defined string procedure, the CompleteItemString method first concatenates selection markers to each end of the string (for multiple choice lists only). Then it concatenates window background characters to each end–LeftPad characters on the left side and RightPad on the right.

For a newly initialized PickList, LeftPad and RightPad are zero.

The length of these padding strings must be counted in the total item width passed to PickList constructors and to the ChangeItemWidth method.

Example

```
PizzaTop.SetPadSize(5, 5);
```

Adds a five blank spaces to the left and right of the item string.

See Also

ChangeItemWidth
RawWindow.SetBackChar
SetSelectMarker

SetPickAttr

Declaration

```
procedure SetPickAttr(ItemType : pkItemType; Selected : Boolean;
                      Color, Mono : Byte);
```

Purpose

Set one group of colors--disables flexwriting.

Description

ItemType, which specifies the category of item to which these attributes apply, is one of pkNormal, pkAlternate, or pkProtected. If Selected is True, the attributes will apply when the selection bar is positioned over the item, otherwise when the item is not current. Note that the combination of ItemType=pkProtected and Selected=True can never occur. Color and Mono are the video attributes to use in color and monochrome video modes, respectively.

In order to support flexwriting, each combination of ItemType, Selected, and Color/Mono can hold up to four video attributes. SetPickAttr initializes each of the four bytes to the same specified value, effectively disabling flexwriting for this item type. Note, however, that SetPickAttr does not reset the internal flag that PickList uses to decide whether to call FastWrite or FlexWrite when writing to the screen.

Example

```
PizzaTop.SetPickAttr(pkNormal, True, $02, $70);
PizzaTop.SetPickAttr(pkNormal, False, $03, $07);
```

Sets the normal, selected attribute to cyan on black for color monitors, reverse video for monochrome monitors. Sets the normal, unselected attribute to green on black for color monitors, normal video for monochrome monitors.

See Also

SetPickFlex

Declaration

```
procedure SetPickFlex(ItemType : pkItemType; Selected : Boolean;
                      var Color, Mono : FlexAttrs);
```

Purpose

Set one group of colors for flexwriting.

Description

SetPickFlex works like SetPickAttr except that it transfers the four unique bytes specified by each FlexAttrs array into the PickList's internal storage. It also sets an internal flag that causes PickList to use FlexWrite rather than FastWrite when writing to the screen. When this flag is set, the characters <CtrlA>, <CtrlB>, and <CtrlC> are interpreted to activate the flexible attributes and cannot be written literally.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick;
const
  NumPizzaToppings = 5;
  SelColorFlex : FlexAttrs = ($1E, $1F, $07, $07);
  SelMonoFlex  : FlexAttrs = ($07, $70, $07, $07);
  UnsColorFlex : FlexAttrs = ($01, $03, $07, $07);
  UnsMonoFlex  : FlexAttrs = ($70, $07, $07, $07);
var
  PizzaTop : PickList;

  {$F+}
  procedure PizzaTopping(Item : Word; Mode : pkMode;
                         var IType : pkItemType; var IString : String;
                         PickPtr : PickListPtr);
    {-Sets item string and item options (if any)}
  begin
    case Item of
      1 : IString := ' '^A'P'^A'epperoni';
      2 : IString := ' '^A'S'^A'ausage';
      3 : IString := ' '^A'M'^A'ushrooms';
      4 : IString := ' '^A'A'^A'nchovies';
      5 : IString := ' '^A'O'^A'nions';
    end;
  end;

begin
  PizzaTop.InitCustom(20, 5, 60, 10, DefaultColorSet,
                      DefWindowOptions or wBordered,
                      15, NumPizzaToppings, PizzaTopping,
                      PickVertical, SingleChoice);
  PizzaTop.SetPickFlex(pkNormal, True, SelColorFlex, SelMonoFlex);
  PizzaTop.SetPickFlex(pkNormal, False, UnsColorFlex, UnsMonoFlex);
  PizzaTop.Process;
  PizzaTop.Done;
end.
```

This is a complete program showing how flexwriting can enhance the appearance of a pick list. For color monitors we use blue on black for unselected items, but with the first letter in cyan. Selected items are shown in yellow on blue with the first letter in white on blue. Note the use of the <CtrlA> characters in the item strings to activate the first special flex attribute.

See Also

FlexWrite(OPCRT) SetPickAttr

SetRowLimits / SetSearchMode / SetSearchStart

Declaration

```
procedure SetRowLimits(MinRows, MaxRows : Word);
```

Purpose

Set values for min and max rows.

Description

The method is appropriate only for multi-column lists. MinRows specifies the minimum number of items in each column and MaxRows the maximum. The values don't affect the size of the pick window; they control the configuration of the list within the window.

See the introductory section "Window Sizing for Pick Lists" for more information and an example.

Declaration

```
procedure SetSearchMode(SF : pkSearchFunc);
```

Purpose

Set a different search mode.

Description

The address of the function SF is stored in an internal data field. The next time Process is called and an alphanumeric character is entered, Process will call the specified routine to search for a matching item.

OPPICK interfaces several routines for this purpose. If you call SetSearchMode, you'll want to specify one of the following: PickNoSearch, PickStringSearch, PickAltStringSearch, PickCharSearch, PickCharExit, or PickAnyCharExit. A newly initialized PickList uses the PickNoSearch function, which doesn't do any searching at all.

See the discussion of "String Searching" in the introduction to this section for more information.

Example

```
PizzaTop.SetSearchMode(PickCharExit);
```

Sets PizzaTop to select an item and exit if a key pressed matches the first letter of an item.

Declaration

```
procedure SetSearchStart(SStart : Byte);
```

Purpose

Set start offset for string searching.

Description

When alphanumeric string searching is enabled, the search engine doesn't necessarily start looking at the first character of each item string. It does so by default, but you can call SetSearchStart to make it search from a different position. This may be useful if all the strings start with a similar prefix.

See the discussion of "String Searching" in the introduction to this section for more information.

Example

```
PizzaTop.SetSearchStart(2);
```

Sets PizzaTop to start searching on the second character of each topic.

Declaration

```
procedure SetSelectMarker(Left, Right : String);
```

Purpose

Define strings to mark selected items.

Description

The method is appropriate only for multiple choice lists. The specified Left and Right strings are attached to each selected item string returned by the user string procedure.

For a newly initialized PickList, Left and Right are empty strings. Hence, you won't be able to detect selected items in a multiple choice list unless you call SetSelectMarker (or you take steps to display the selected items in an alternate video attribute).

The length of these select markers must be counted in the total item width passed to PickList constructors and to the ChangeItemWidth method.

Example

```
PizzaTop.SetSelectMarker('<', '>');
```

When using the MultipleChoice command handler, selected items will be enclosed in angle brackets.

See Also

CompleteItemString

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a list in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the PickList to the stream S. It starts by calling CommandWindow.Store, then continues to write the data specific to a PickList.

To check for errors that may have occurred, call the stream's GetStatus function.

The stream registration procedure for a PickList is PickListStream. There are also a number of procedure pointers that need to be registered independently. The example program below demonstrates how. These registration calls take a little extra effort on your part; in return, Turbo's smart linker is able to strip out the code sections of OPPICK–alternate orientations, alphanumeric search routines, and so on–that your program doesn't actually use. If we registered all of them for you, the linker would hook them all in.

The list's orientation routine (PickVertical, PickHorizontal, or PickSnaking) must be registered. Failing to do so will result in the error epFatal+ecBadPickOrient when the stream is loaded.

The list's command handler (SingleChoice or MultipleChoice) must also be registered. Failing to do so will result in the error epFatal+ecBadPickHandler when the stream is loaded.

If the list has a user-defined string procedure (as opposed to overriding the ItemString virtual method), that procedure must be registered. Failing to do so will result in runtime error 211 (attempt to execute an abstract procedure) when the pick list is used.

If the list has a user-defined move procedure, it should be registered. Otherwise the move procedure won't be activated when the list is reloaded.

If the list enables alphanumeric searching, the search function (PickStringSearch, PickAltStringSearch, PickCharSearch, PickCharExit, or PickAnyCharExit) should be registered. Otherwise searching will be disabled when the stream is loaded.

If the PickList uses any but the default CommandProcessor (PickCommands), the address of that object must be registered. See CommandWindow.Store for an example.

Example

```
uses  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick;
const NumPizzaToppings = 5;
var   PizzaTop : PickList;
      S : BufIdStream;

  {$F+}
  procedure PizzaTopping(Item : Word; Mode : pkMode;var IType : pkItemType;
                         var IString : String; PickPtr : PickListPtr);
    {-Sets item string and item options (if any)}
  begin
    case Item of
      1 : IString := ' Pepperoni';
      2 : IString := ' Sausage';
      3 : IString := ' Mushrooms';
      4 : IString := ' Anchovies';
      5 : IString := ' Onions';
    end;
  end;
  {$F-}
```

```
begin
  PizzaTop.InitCustom(35, 5, 45, 10, DefaultColorSet,
                      DefWindowOptions or wBordered,
                      20, NumPizzaToppings, PizzaTopping,
                      PickVertical, SingleChoice);
  PizzaTop.SetSearchMode(PickCharSearch);
  PizzaTop.wFrame.AddShadow(shBR, shOverWrite);
  PizzaTop.wFrame.AddHeader(' PICK ', heTC);

  {Store PizzaTop to a stream}
  S.Init('PIZZA.STM', SCreate, 1024);
  S.RegisterHier(PickListStream);
  S.RegisterPointer(ptPickVertical, @PickVertical);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.RegisterPointer(ptPickCharSearch, @PickCharSearch);
  S.RegisterPointer(1000, @PizzaTopping);
  S.Put(PizzaTop);
  S.Done;
  PizzaTop.Done;
end.
```

Instantiates PizzaTop, adds a few features, then stores PizzaTop to a stream. The example registers the addresses of the list orientation routine (PickVertical), the command handler (SingleChoice), the search function (PickCharSearch), and the string procedure (PizzaTopping). Note the use of the '@' operator to return a pointer to each named procedure or function. Each of the standard support routines provided by OPPICK has a predefined registration code that should be used to avoid conflicts:

Procedure or Function	Pointer Registration Code	Category
PickNoSearch	ptPickNoSearch	String searching
PickStringSearch	ptPickStringSearch	"
PickAltStringSearch	ptPickAltStringSearch	"
PickCharSearch	ptPickCharSearch	"
PickCharExit	ptPickCharExit	"
PickAnyCharExit	ptPickAnyCharExit	"
PickVertical	ptPickVertical	List orientation
PickHorizontal	ptPickHorizontal	"
PickSnaking	ptPickSnaking	"
SingleChoice	ptPickSingleChoice	Command handling
MultipleChoice	ptPickMultipleChoice	"
NoMoveAction	ptPickNoMoveAction	Move procedure
NoPickString	ptNoPickString	String procedure

We've registered the string procedure (PizzaToppping) with a code of 1000. Any value 1000 or larger that is unique to this stream would be acceptable. A program that loads this stream must use the same code.

See the Load example for a program to load this stream.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

CommandWindow.Store Load

UpdateContents / WhereSelect / WhereSelectAbs

Declaration
```
procedure UpdateContents; virtual;
```

Purpose
Redraw the complete pick window.

Description
You shouldn't need to call UpdateContents directly. Call Draw instead.

UpdateContents performs the following operations:

It allocates the select BitSet for multiple choice pick lists if it has not already been allocated.

If the pick list has no items, UpdateContents clears the window and exits.

It recomputes the list configuration in case the number of items, item width, list orientation, etc., have changed since the last call. To save time, UpdateContents skips this step if an internal flag is set to indicate that the Process method is active. (The list configuration should never be changed while Process is active.)

It completely redraws the items currently visible in the window and highlights the currently selected item.

It redraws the more header, if any, and updates the scroll bars.

See Also
RawWindow.Draw StackWindow.Draw

Declaration
```
procedure WhereSelect(var X, Y : Byte);
```

Purpose
Return window-relative coordinates of selected item.

Description
If the item is padded, the coordinates returned here won't actually specify where the text of the item starts. However, these are the coordinates the PickList itself uses to draw the string returned by CompleteItemString. The hardware cursor is positioned at the same location, although it usually isn't visible.

You might use this method to create special effects, like drawing a flashing box around the selected item after Process exits.

Example
```
PizzaTop.Process;
PizzaTop.WhereSelect(X, Y);
PizzaTop.wChangeAttribute(20, Y, X, $02);
Delay(1000);
PizzaTop.Erase;
```

Changes the selected item's attribute for one second, then erases the window.

Declaration
```
procedure WhereSelectAbs(var X, Y : Byte);
```

Purpose
Return absolute coordinates of selected item.

Description
Works just like WhereSelect except that it returns absolute screen coordinates, not window-relative ones.

E. OPDIR - Directory and Path Lists

The OPDIR unit implements two objects derived from the PickList: one, the DirList, is for selecting DOS filenames; the other, PathList, is for selecting subdirectory names. These objects inherit all of the flexibility of the window and pick list objects, and add many new features for dealing with the DOS directory structure:

- wildcard searches for filenames
- selection of files by attribute
- selection of files by user-definable criteria
- user-definable sort order
- user-definable name format, including size, date, time, and attribute
- specialized header for filename search mask
- support for multiple file marking
- graphic tree format for path list display

This section is divided into two parts. The first covers the DirList, and the second the PathList.

1. DirList: Filename Selection

DirList can select a single file at a time or, by building on PickList's multiple choice capabilities, mark many files for a followup operation. The DESKPOP demonstration program shows both of these techniques: whenever it prompts for a filename to edit or browse, you can use wildcards to get a single choice DirList window; the file manager window uses multiple choice selection so you can copy, move, or delete groups of files in one pass.

Not everyone needs to write a DOS file manager, but almost all of us need to prompt for filenames. Let's start off with a working demonstration of what's required to use the DirList in this way.

```
program DirectoryListExample;  {EXDIR.PAS}
uses
  Dos, OpString, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  Finished : Boolean;
begin
  if not Dir.InitCustom(25, 5, 55, 20,                {Window coordinates}
                        DefaultColorSet,              {ColorSet}
                        DefWindowOptions or wBordered, {Window options}
                        MaxAvail,                     {Heap space for files}
                        PickVertical,                 {Pick orientation}
                        SingleFile)                   {Command handler}
  then begin
    WriteLn('Failed to Init DirList,  Status = ', InitStatus);
    Halt;
  end;

  {Set desired DirList features}
  Dir.SetPosLimits(1, 1, ScreenWidth, ScreenHeight-1);
  Dir.SetPadSize(1, 1);
  Dir.diOptionsOn(diOptimizeSize);
  Dir.AddMaskHeader(True, 1, 30, heTC);
  Dir.SetSortOrder(SortName);
  Dir.SetNameSizeTimeFormat('<dir>', 'Mm/dd/yy', 'Hh:mmt');
  Dir.SetMask('*.*', AnyFile);
```

```
    {Pick a file}
    Finished := False;
    repeat
      Dir.Process;
      case Dir.GetLastCommand of
        ccSelect :
          FastWrite(Pad('Selected: '+Dir.GetSelectedPath, ScreenWidth),
                    ScreenHeight, 1, TextAttr);
        ccError  :
          FastWrite('Error '+Long2Str(Dir.GetLastError),
                    ScreenHeight, 1, TextAttr);
        ccQuit   :
          Finished := True;
        {Process other exit commands}
      end;
    until Finished;
    Dir.Done;
  end.
```

First we call the DirList's constructor to set the window coordinates, colors, and options. Three of the constructor's parameters are specific to the DirList.

We must specify how much heap space to use for storage of the filenames. The DirList can't predict how many files it will need to store so it just allocates a fixed size chunk of heap and uses that. The amount of heap space needed for each file depends on the display format, but in this example it works out to 80 bytes per file (this is the worst case). Although we ask for the largest single block of heap space (MaxAvail), DirList will use at most a 64K block. If that much heap is available, the example can manage up to 813 files at a time.

As with PickList, we must select a list orientation. Only one column of names fits in the window, so the orientation is moot in this case. We can also provide single or multiple file selection; here we only want one at a time, so we specify SingleFile.

We have a vast collection of customization options to choose from. This example uses SetPosLimits (inherited from RawWindow) to keep the DirList from expanding onto the last line of the screen, which we'll use for a status line. SetPadSize (from PickList) pads each filename with spaces. diOptionsOn enables DirList's diOptimizeSize option, which automatically resizes the directory window to fit the number of files. AddMaskHeader enables a specialized frame header that displays the current directory search mask. SetSortOrder tells DirList to sort the filenames into alphabetical order. SetNameSizeTimeFormat activates a detailed display format, including name and extension, size, date, and time. Note that we can specify picture masks for the date and time, making international support easier.

Finally we specify the search mask, '*.*', and the file attribute, AnyFile. In a real application, of course, we'd prompt the user for a name, but that's more than we want to show here.

After that we use the DirList just like a PickList by calling Process and then GetLastCommand. If a selection was made, we display the selected pathname by using the GetSelectedPath method. When dealing with files, it's very important to check for errors; the case label ccError does that here.

The DirList provides a method that's even more convenient than Process for getting filenames. It's called GetFileName; be sure to read about it in the methods that follow.

Conditional Defines

OPDIR is affected by the same conditional defines as its antecedents, plus one more. The defines we've already covered are UseMouse, UseScrollBars, UseHotSpots, UseShadows, and UseAdjustableWindows.

The new one is UseDates and it controls whether OPDIR uses the OPDATE unit. If you want to use any of the filename formats that include date and time strings, you'll need to activate this define by assuring that OPDEFINE.INC has a line like the following:

```
{$DEFINE UseDates}
```

If you don't need dates and times in OPDIR (or in any of the other units, like OPENTRY, that your program uses), deactivating this define will reduce the memory usage of the compiler and your program.

Directory Commands

The commands in DirList work just like those in PickList, with a couple of exceptions. When the ccSelect command is processed by a DirList and the current item is a subdirectory name, the Process method doesn't exit. Instead, the DirList adjusts its internal masks to point to the new directory, finds matching files there, and displays a new list of matching files. Similarly, in a multiple choice DirList, Process won't allow a subdirectory name to be marked as a choice; it either ignores the attempt to select it or displays the new subdirectory's files.

There is another point to be made about multi-file selections in a DirList: the selection set is only capable of storing marked files in one directory at a time. If you mark a group of files and then press <Enter> while the highlight bar is on a subdirectory, DirList will clear the selection set and display an unmarked list of files in the new subdirectory. Returning to the prior directory will not restore the marked list. Managing selection sets for multiple directories requires far too much overhead to be worthwhile; and besides, it raises a number of questions about how applications should act when they have these selection sets to work with.

By default, the DirList points to the same CommandProcessor instance as does the PickList. It's important to keep this in mind if your program uses DirLists and PickLists independently and it needs to modify the command mapping of one but not the other. In that case the best approach is to define a separate CommandProcessor and key array for the DirList. You can start out by copying PickList's KeySet constant, which you'll find in OPPICK.ICD, and renaming a few identifiers:

```
const
  {Keystroke to command mapping}
  DirKeyMax       = 160;
  DirKeyID        : string[10] = 'opdir keys';
  DirKeySet       : array[0..DirKeyMax] of Byte = (
  {length keys         command type      key sequence}
  3,      $00, $48,    ccUp,             {Up}
  3,      $00, $50,    ccDown,           {Down}
  ...
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0
  );
```

If you never plan to use some of the commands, like those for multiple file selection, you can cut them out of the list to save some space.

Next you need to declare a variable of type CommandProcessor and to initialize it (prior to initializing an instance of a DirList):

```
var
  DirCommands : CommandProcessor;
...
DirCommands.Init(@DirKeySet, DirKeyMax);
```

If you want to dynamically add or modify commands, you will call

```
DirCommands.AddCommand(...);
```

Finally, after initializing each DirList, you set its CommandProcessor:

```
Dir.SetCommandProcessor(DirCommands);
```

Filename Filtering

Whenever your application specifies a new search mask and calls DirList.Process, the DirList goes through a series of steps which can be outlined as follows:

- call DOS.FindFirst and DOS.FindNext to find the files
- filter out unwanted files
- sort the remaining files
- created a formatted display string for each file
- optionally select the initial choice for the PickList
- optionally optimize the window size
- optionally update the window's mask header

In this section we'll describe the second step, filtering unwanted files.

In DirList's actual implementation, the first and second steps are combined. Each time FindFirst (or FindNext) returns a file, it's checked for acceptability before it is added to an internal array.

As you probably know, the FindNext routine returns information about each file in a record the DOS unit calls a SearchRec. OPDIR copies a subset of this information into another record called a DirRec, which has the following declaration:

```
DirRec          =           {Data from DOS SearchRec}
  record
    Attr        : Byte;     {File attribute}
    Time        : LongInt;  {Date/time stamp}
    Size        : LongInt;  {Size of file}
    Name        : FileStr;  {Name and extension of file, 12 characters max}
  end;
```

To check a particular DirRec, OPDIR calls the virtual method Reject:

```
function Reject(var X : DirRec) : Boolean; virtual;
  {-Return True to exclude a particular entry from pick}
```

When this function returns True, DirList ignores the associated file as if it didn't exist. The function can use any of the information in the DirRec to make its decision: for example, whether the file's extension is acceptable or its date falls in a certain range.

DirList's Reject function delegates its authority by calling another user-defined routine via a procedure pointer (hence following the same strategy outlined in the introduction to OPPICK). You can specify a routine to take charge with DirList's SetRejectFunc method:

```
procedure SetRejectFunc(RF : diRejectFunc);
  {-Set alternate reject function for file selection}
```

The parameter to SetRejectFunc is itself a function that has been compiled using the FAR model, is global, and matches the following prototype declaration:

```
{$F+}
function MyRejectFunc(var X : DirRec; D : DirListPtr) : Boolean;
```

The extra parameter D passed to this function is the address of the DirList object that called it, thereby allowing the reject function to use the object's methods and internal data if needed.

The OPDIR unit interfaces three prewritten reject functions that are useful in common situations:

```
function RejectNone(var D : DirRec; DirPtr : DirListPtr) : Boolean;
  {-Don't reject any files}
function RejectExtensions(var D : DirRec; DirPtr : DirListPtr) : Boolean;
  {-Reject files with extensions in RejectSet}
function RejectFiles(var D : DirRec; DirPtr : DirListPtr) : Boolean;
  {-Reject all but subdirectories}
```

The first one, RejectNone, is the default. This function returns False in all but two cases: first, when the file is named '.', which is DOS's way of denoting the current directory but not very useful in a pick list; and second, when the file's attribute is VolumeID, since we don't want someone to select that special "file."

RejectExtensions rejects in addition any files whose extensions are in a predefined set. For example, you might not want to display EXE, COM, or BAK files. DirList provides a method for specifying the extensions:

```
procedure AddRejectString(S : String);
  {-Add a string to the set to reject}
```

Call this method once for each extension to be rejected. To make the introductory example reject EXE, COM, and BAK files, we'd add the following lines to the "customization" section of the code:

```
{Set desired DirList features}
...
Dir.AddRejectString('EXE');
Dir.AddRejectString('COM');
Dir.AddRejectString('BAK');
Dir.SetRejectFunc(RejectExtensions);
```

Note that AddRejectString isn't limited to storing just file extensions. It uses a StringSet object to store an arbitrary set of strings and it's up to the reject function to decide how to interpret them. If you need to reject files based on something besides their extensions, take a look at the RejectExtensions function (in OPDIR.IN1) and you'll easily be able to write a similar routine.

RejectFiles is used internally by DirList's ChangeDirectory method. It causes the directory list to show *nothing but* subdirectory names: no regular files at all. This function is needed because DOS's file attribute checking provides no way to get the same result directly.

Sorting the List

FindFirst, FindNext, and the Reject function build an array of files to be displayed. The array is arranged in the order that DOS provided it. At first, this order matches the sequence in which the files were created. As soon as files are deleted, however, DOS reuses the freed-up directory slots when new files are created. Picking a file from such a list is neither intuitive nor efficient.

The DirList therefore provides the opportunity to sort the list of files. It calls the virtual method SortList to do so. SortList uses **Object Professional**'s standard strategy: it calls a user-defined procedure through a procedure pointer to do the real work. You specify such a procedure with DirList's SetSortOrder method:

```
procedure SetSortOrder(SP : diSortProc);
  {-Set alternate routine for sorting}
```

The parameter to SetSortOrder is itself a procedure that has been compiled using the FAR model, is global, and matches the following prototype declaration:

```
{$F+}
procedure MySortProc(D : DirListPtr);
```

The parameter D passed to this procedure is the address of the DirList object that called it, thereby allowing the sort procedure to use the object's methods and internal data as needed.

OPDIR interfaces several sort procedures that will serve for most situations:

```
procedure SortNone(DirPtr : DirListPtr);
  {-No sort}
procedure SortName(DirPtr : DirListPtr);
  {-Sort alphabetically by name, then by extension}
procedure SortDirName(DirPtr : DirListPtr);
  {-Sort directories first, then alphabetically by name and extension}
procedure SortSize(DirPtr : DirListPtr);
  {-Sort by size, then by name and extension}
procedure SortTime(DirPtr : DirListPtr);
  {-Sort by date, then by name/extension}
```

SortNone is the default; it does nothing at all. The resulting directory list will appear in the same order as the DOS DIR command.

SortName sorts the files and directories in strict alphabetical order. You've seen an example of this sort in the introductory example. It was activated with the statement:

```
Dir.SetSortOrder(SortName);
```

SortDirName puts the files in the order that we prefer: the subdirectories first, in alphabetical order, followed by the normal files in alphabetical order. The DirList has another option that cooperates with this sort: diSetFirstFile, when enabled, causes DirList to position the initial highlight bar just past the directories.

SortSize puts the files in order of ascending size and SortTime in order of ascending date/time.

All of the sorts share a common recursive quicksort engine which is fast and efficient. OPDIR enables stack checking within the quicksort. If the recursive sort should ever run out of stack space (which is unlikely), you'll get a runtime error 202 within the OPDIR.IN1 source file.

If you need to write a sort procedure that isn't already covered, study the sort routines in OPDIR.IN1. For efficiency, they use the object's internal data structures and methods, but they remain straightforward nonetheless.

Filename Formatting

After building and sorting the array of DirRec, the DirList initializes a separate array containing ready-to-display strings for each of the files. You might think that the strings could be computed on the fly as OPPICK asked for them. Experience proves that this approach leads

to horribly sluggish performance, especially when formatted date and time strings are part of the result. Therefore we decided that it was worth the memory cost to preformat each string.

Again OPDIR uses the virtual method/procedure pointer strategy to offer all the flexibility you need for formatting these strings. The virtual method is called Format:

```
procedure Format(var X : DirRec; var pkCat : Byte; var S : String); virtual;
    {-Return a formatted string and category for a directory entry}
```

DirList calls this method for each element in the DirRec array. Format returns a string, S, to display and a pick category, pkCat, that OPPICK uses to determine a color for the item. pkCat must equal pkNormal, pkAlternate, or pkProtected, as defined by OPPICK.

Format in turn calls a global, FAR procedure whose declaration matches the following:

```
{$F+}
procedure MyFormatProc(var X : DirRec; var pkCat : Byte;
                       var S : String; D : DirListPtr);
```

As before, the parameter D allows the procedure to identify the object that called it and to use that object's data and methods if needed. About the only method MyFormatProc might be interested in is GetSelectedDir, which returns the drive and directory where the files were found.

OPDIR interfaces a number of standard formatting procedures. Instead of referring to them directly, however, you inform the DirList of your formatting needs by calling one of the following methods:

```
procedure SetNameFormat;
  {-Initialize for compact nnnnnnnn.xxx format}
procedure SetNameSizeFormat(DirStr : String);
  {-Initialize for nnnnnnn.xxx ssssssss format}
procedure SetNameSizeKFormat(DirStr : String);
  {-Initialize for nnnnnnn.xxx ssssssk format}
{$IFDEF UseDates}
procedure SetNameTimeFormat(DatePicture, TimePicture : String);
  {-Initialize for name date time format}
procedure SetNameSizeTimeFormat(DirStr, DatePicture, TimePicture : String);
  {-Initialize for name size date time format}
procedure SetNameSizeKTimeFormat(DirStr, DatePicture, TimePicture : String);
  {-Initialize for name size date time format}
procedure SetAllFormat(DirStr, DatePicture, TimePicture : String);
  {-Initialize for name size date time attr format}
{$ENDIF}
procedure SetUserFormat(MaxFmatLen : Byte; FF : diFormatProc);
  {-Set up for a user-defined format function}
```

These methods store not only the appropriate procedure variable for the Format method to call, but also store certain picture strings so dates and times can be formatted correctly.

DirStr is the string to display in the list's "size" column when the item is a subdirectory. Although this parameter is passed as a string, the DirList stores at most 9 characters: enough to hold 'directory' or '<dir>'. Also note that this string must fit in the same column with the sizes. In 'K' (kilobytes) format, the size column is only 7 characters wide.

DatePicture and TimePicture follow the rules for picture masks described in the OPDATE unit. Here are a few examples appropriate to directory listings:

```
Picture                 Result
-----------------------------------
'mm/dd/yy'              '01/02/80'
'MM/dd/yy'              ' 1/02/80'
'DD/mm/yyyy'            ' 2/01/1980'
'dd-NNN-yyyy'           '02-JAN-1980'
'hh:mm:ss'              '13:02:00'
'hh:mm'                 '13:02'
'hh:mm te'              '01:02 pm'
'Hh:mmt'                ' 1:02p'
```

The SetUserFormat method provides for a completely user-defined format. MaxFmatLen specifies the maximum string length returned by the format procedure. It's essential that the procedure never return a string longer than this. If it does it will overwrite memory that it doesn't own, causing potentially disastrous side effects. FF is the format procedure that DirList will call for each entry.

To provide an example of a user-defined format procedure, let's assume that we want a directory listing that shows just the filenames (no extensions) plus the size in kilobytes of each file. First, we compute the maximum length of each string:

```
0        1
12345678901234
--------------
nnnnnnnn__sssk
```

Assuming that the file sizes will never exceed 999k bytes (or about 1 megabyte), each string is no longer than 14 characters.

Now we can write the format routine itself:

```
{$F+}
procedure CustomFormat(var X : DirRec; var pkCat : Byte;
                       var S : String; D : DirListPtr);
var
  Siz : LongInt;
begin
  if X.Attr and Directory = Directory then begin
    S := X.Name;                              {Return the directory name}
    pkCat := pkAlternate;                     {Return alternate category}
  end else begin
    Siz := (X.Size+1023) div 1024;            {Convert to kBytes}
    if Siz > 999 then
      S := '***'                              {Handle overflow}
    else
      Str(Siz:3, S);                          {Convert number to string}
    S := Pad(JustName(X.Name), 8)+'  '+S+'k'; {Build the final string}
    pkCat := pkNormal;                        {Return normal category}
  end;
end;
{$F-}
```

CustomFormat first separates the directories, which always have zero size, from the real files. It classifies the directories in the pkAlternate category so that the pick list manager will display them in a different color. For files, CustomFormat first converts the actual file size (X.Size) to kilobytes, being careful not to truncate partial kilobytes. It plays safe by displaying asterisks if the size does exceed 999k; otherwise it converts the numeric size to a right-justified string. Then

it concatenates the file name and the size. Finally it sets pkCat to indicate a normal pick item. You'll note that this routine uses various functions from OPSTRING.

Activate this custom formatting routine with the following call:

```
Dir.SetUserFormat(14, CustomFormat);
```

The fact that we're not showing the extensions of each file probably means that we want to select just a particular extension, perhaps 'DOC' or 'WKS'. To do so, pass a search mask like '*.DOC' or '*.WKS' to the SetMask method.

Constants

```
diDirsUpcase      = $0001;  {Raise directory names to uppercase}
diFilesUpcase     = $0002;  {Raise filenames to uppercase}
diExitIfOne       = $0004;  {Autoselect in GetFileName if one matching file}
diSetFirstFile    = $0008;  {Position select bar on first non-directory}
diOptimizeSize    = $0010;  {Optimize window size when directory is read}
diUserNewHdr      = $0020;  {Set for user move routine when header is changed}
```

These are the options that affect the behavior of a DirList. diDirsUpcase forces directory names to uppercase. If this option is not enabled, directory names are forced to lowercase. Note that the case change occurs immediately after the name is obtained from FindFirst/FindNext; the Format routine can still override the case. diFilesUpcase controls the case of non-directory files.

diExitIfOne modifies the behavior of a single DirList method, GetFileName. When diExitIfOne is enabled and the search mask passed to GetFileName contains no wildcard characters, GetFileName will exit without displaying a pick window and without requiring further keyboard input if exactly one file matches the mask. See GetFileName for more information.

If diSetFirstFile is enabled, Process positions the pick list selection bar over the first non-directory entry whenever it reads a new list of files. Otherwise it puts the bar on the first entry.

diOptimizeSize enables automatic window sizing, so that the pick window shrinks and grows automatically depending on the number of matching files. The resizing is performed by PickList's OptimizeSize method; see that entry for more details. Note that this option works only if the UseAdjustableWindows define is active.

diUserNewHdr is set by DirList whenever it updates the search mask header to indicate a new directory. See the AddMaskHeader method for more information.

```
DefDirOptions     : Word = diSetFirstFile;
```

The default options for a DirList.

```
BadDirOptions     : Word = diUpdating+diUserNewHdr+diReadDir+
                           diUpdateHdr+diResize+diReposition+diResort+
                           diReformat+diFullPathHdr;
```

Options used internally by DirList. The diOptionsOn and diOptionsOff methods will not change these bits of a DirList option word.

```
diReadOnlyChar    : string[1] = 'R'; {Characters to display file attributes}
diHiddenChar      : string[1] = 'H';
diSystemChar      : string[1] = 'S';
diArchiveChar     : string[1] = 'A';
```

When SetAllFormat has been called to enable the most detailed filename display, these strings determine how OPDIR displays the file attributes. For each bit set in the file's attribute byte, OPDIR concatenates the appropriate string. To disable the display of a particular bit, just set the corresponding string to an empty string.

Types

```
DispRec          =              {Variable length record for display entry}
  record
    Cat          : Byte;        {Pick category}
    Disp         : String;      {String to display--partially allocated}
  end;
DispRecPtr       = ^DispRec;    {Pointer to a DispRec}
```

OPDIR manages an array of DispRec records; one for each non-rejected file. The Disp field only uses as much space as the selected format requires.

```
FileStr          = String[12]; {Room for just a filename}
DirRec           =             {Data from DOS SearchRec}
  record
    Attr         : Byte;       {File attribute}
    Time         : LongInt;    {Date/time stamp}
    Size         : LongInt;    {Size of file}
    Name         : FileStr;    {Name and extension of file}
  end;
DirRecPtr        = ^DirRec;
DirRecArray      = array[1..($FFF1 div SizeOf(DirRec))] of DirRec;
DirArrayPtr      = ^DirRecArray;
```

OPDIR also manages an array of DirRec. It copies the record returned by each call to FindFirst or FindNext into a DirRec. If the DirRec is not rejected by the reject function, it stays in the array. User-defined reject, sort, and format routines must refer to the fields of a DirRec. These fields are exactly equivalent to those defined in the SearchRec type in Turbo Pascal's DOS unit. For more information see the Borland Turbo Pascal Reference Guide.

Note that each DirList object combines a DirRecArray and an array of DispRec in a single 64K segment. This sets a limit on the maximum number of files it can display at once. The FileCapacity method returns the actual limit for a specific situation. With a 64K heap buffer, a DirList manages about 500-2000 files, depending on the display format.

```
diRejectFunc     = function (var X : DirRec; D : DirListPtr) : Boolean;
diSortProc       = procedure (D : DirListPtr);
diFormatProc     = procedure (var X : DirRec; var pkCat : Byte;
                              var S : String; D : DirListPtr);
```

Types that define valid procedures and functions for user-defined reject, sort, and format routines. See the introduction for more information.

```
diInitCommandProc = procedure (D : DirListPtr;
                               var StringProc : pkStringProc;
                               var CommandInitProc : pkGenlProc);
```

A procedure type used by DirList constructors to specify a single or multiple choice directory list. OPDIR interfaces the SingleFile and MultipleFile procedures that match this type.

```
DirListPtr = ^DirList;
DirList =
  object(PickList)
    ...
  end;
```

The DirList object.

Declaration

```
procedure AddMaskHeader(FullPath : Boolean; PadSize, MaxLen : Byte;
                        Posn : HeaderPosType);
```

Purpose

Add a specialized header to display the search mask.

Description

AddMaskHeader makes a call to wFrame.AddHeader after it initializes internal data structures to keep track of the mask header. DirList will then automatically update it to display the current search mask whenever it is changed.

If FullPath is True, the header will show the drive and directory as well as the search mask itself.

PadSize specifies the number of spaces to add to each end of the header before displaying it. MaxLen is the total number of characters to reserve for the header, including any padding. If the actual mask would exceed this length, DirList will remove internal characters from the header before displaying it. In order to provide a meaningful display, AddMaskHeader always makes sure that MaxLen-2*PadSize is at least six characters.

Posn specifies the frame-relative position of the header (heTL, heTC, heTR, heBL, heBC, or heBR).

Whenever DirList updates the mask header, it sets a flag in the internal DirList options word. A user-defined move procedure (see PickList.PreMove) can take advantage of this flag to perform a special action–updating a file and byte count status line, for example–whenever a new directory appears. It is the user routine's responsibility to reset the flag. Here's a sequence of code that would appear in the move procedure to take advantage of this flag:

```
if DirListPtr(P)^.diOptionsAreOn(diUserNewHdr) then begin
  {User is viewing a different directory}
  ...
  {Reset the flag}
  diOptionsOff(diUserNewHdr);
end;
```

Example

```
Dir.SetMask('*.*', AnyFile);
Dir.AddMaskHeader(True, 1, 20, heTC);
```

If the current directory were 'C:\TEST', then the directory window would display the header ' C:\TEST*.* ' centered in the top border. If the user navigated to another directory, DirList would update the header accordingly. If the user got to a directory named C:\TEST\LONGDIR\NEXTDIR, then DirList would shorten the header to read ' C:...R\NEXTDIR*.* '. Note that the length of the header is 20 characters.

See Also

SetMask

AddRejectString

Declaration

```
procedure AddRejectString(S : String);
```

Purpose

Add a string to the set to reject.

Description

DirList manages an internal StringSet object. You can use this StringSet to reject files based on their extensions or perhaps other characteristics. In combination with the RejectExtensions reject function, the internal StringSet is interpreted specifically as a set of extensions. In this case, the parameter S should be a particular extension ('BAK', for example) that you don't want to appear in the directory list.

Because AddRejectString must allocate heap space to store the specified string and supporting data structures, it's possible for this routine to fail with the error epNonFatal+ecOutOfMemory. Call GetLastError to check the status.

For more information, see the discussion of "Filename Filtering" at the introduction to this section.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.AddRejectString('BAK');
if Dir.GetLastError = 0 then
  Dir.SetRejectFunc(RejectExtensions)
else
  {Out of memory} ;
Dir.Process;
```

Reject all files with an extension of 'BAK'.

See Also

ClearRejectStrings StringSet.Add

Declaration

```
procedure ChangeDirectory(Mask : String; ChangeCode : Byte);
```

Purpose

Change to specified directory, or let user browse around to do so.

Description

ChangeDirectory is a shell around DirList's Process method. It is provided to make it easier to use the DirList as a way to change directories. The object that ChangeDirectory references must have already been initialized by calling its constructor. Any customization (reject, sort, format options, etc.) must have already been set up. ChangeDirectory works only with directory lists initialized to pick a single file at a time.

Mask is the name of the directory to change to. If it is a non-empty string and contains no wildcards, ChangeDirectory will attempt to change directly to that directory by calling Turbo Pascal's ChDir procedure, without displaying a pick list at all. If ChDir fails, ChangeDirectory reports the warning epWarning+ecPathNotFound, which you can check by calling GetLastError.

If mask is an empty string or contains wildcards, ChangeDirectory attempts to build a pick list of the matching directories. It does so by temporarily switching to the RejectFiles reject function, which forces DirList to display only subdirectories in its pick list. It converts the empty string to a '*.*' search mask. If no matching directories are found, ChangeDirectory again reports the warning epWarning+ecPathNotFound and exits.

If matching directories are found, ChangeDirectory calls Process to display a pick list of these items. If the user scrolls around this list and presses <Enter> on a certain directory, DirList will use the search mask again to find matching directories in that location.

Here's where the ChangeCode parameter comes in. ChangeCode is the command code for a particular DirList exit command. When the user presses a keystroke combination that maps to this exit command, Process will exit and ChangeDirectory will change the current directory to the last one listed by the pick list. If the user exits with any command besides ChangeCode, ChangeDirectory doesn't change the current directory.

If the DirList window wasn't visible when ChangeDirectory was called, it is erased again before ChangeDirectory exits.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
begin
  if not Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal) then
    Halt;
  Dir.ChangeDirectory('*.*', ccQuit);
end.
```

Display a list of directories and switch to the selected directory when the user presses <Esc> (ccQuit). Pressing <Esc> to change directories is somewhat counter-intuitive. You might either want to assign a different exit key or evaluate PathList.GetPathName for an alternative method of switching directories.

See Also

PathList.GetPathName

ClearRejectStrings / diOptionsAreOn

Declaration
```
procedure ClearRejectStrings;
```

Purpose
Clear the set of strings to reject.

Description
DirList's internal StringSet is cleared and disposed. The entire directory will be reread the next time Process or PreloadDirList is called, in order to avoid missing any files that are no longer rejected.

Example
```
Dir.ClearRejectStrings;
```
Clear any previously defined reject strings.

See Also
AddRejectString StringSet.Done

Declaration
```
function diOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This function tests the directory list options, which are described in the Constant declarations near the beginning of this section.

Example
```
if Dir.diOptionsAreOn(diDirsUpcase) then
  {take some action};
```
Takes action if the diDirUpcase option is on.

See Also
diOptionsOff diOptionsOn

Declaration
```
procedure diOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadDirOptions (see the Constants section, above).

Example
```
Dir.diOptionsOff(diOptimizeSize+diDirsUpcase+diFilesUpcase);
```
Turns off three options at once: window size will not be optimized, directory and file names will be forced to lower case.

Declaration
```
procedure diOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadDirOptions (see the Constants section, above).

Example
```
Dir.diOptionsOn(diOptimizeSize+diDirsUpcase+diFilesUpcase);
```
Turns on three options at once: optimize window size, display both directory and file names in upper case.

Done / FileCapacity

Declaration

```
destructor Done; virtual;
```

Purpose

Dispose of directory list.

Description

This destructor deallocates the heap space that DirList uses to store directory arrays, disposes of its internal StringSet (if any), and then calls PickList.Done.

Example

```
Dir.Done;
```

Disposes of the heap space used by Dir.

See Also

PickList.Done StringSet.Done

Declaration

```
function FileCapacity : Word;
```

Purpose

Return the maximum number of files the list can hold.

Description

When a DirList is initialized, you specify how many bytes of heap space it can use for file storage. This heap space is split between two arrays: an array of DirRec (22 bytes) and an array of DispRec (2 bytes plus the maximum length of each formatted file string). FileCapacity divides the heap space by the bytes per file, and returns the maximum number of files that the DirList can manage.

When a DirList attempts to read a directory holding more files than it can handle, it generates a warning (epWarning+ecTooManyFiles), but will continue processing with the files that it could store.

Example

```
if not Dir.InitCustom(25, 5, 55, 20,                {Window coordinates}
                      DefaultColorSet,              {ColorSet}
                      DefWindowOptions or wBordered, {Window options}
                      MaxAvail,                     {Heap space for files}
                      PickVertical,                 {Pick orientation}
                      SingleFile)                   {Command handler}
then begin
  WriteLn('Failed to Init DirList,  Status = ', InitStatus);
  Halt;
end;
...
Dir.SetNameSizeTimeFormat('<dir>', 'Mm/dd/yy', 'Hh:mmt');
WriteLn('Maximum files: ', Dir.FileCapacity);
```

Assuming that at least 64K of contiguous heap space is available, this example will indicate that 813 can be managed by the DirList. If a simpler display format is used, Dir will be able to store correspondingly more files. For example, using the shorter format activated by SetNameFormat, the list can manage 1477 files.

See Also

Init

Declaration

```
procedure Format(var X : DirRec; var pkCat : Byte; var S : String); virtual;
```

Purpose

Return a formatted string and category for a directory entry.

Description

OPDIR calls this virtual method to convert each raw file record (X) into a formatted string (S) and pick list category (pkCat). The method may be overridden by derived types to implement a specific formatting style. DirList's implementation of Format simply passes on the call to another user-defined format procedure.

See the discussion of "Filename Formatting" near the start of this section for more information and for complete examples.

See Also

SetXxxFormat

Declaration

```
procedure FormatMultiFile(N : Word);
```

Purpose

Rebuild the format string for specified file.

Description

The method calls the format function for the N'th file. It reads the internal DirRec that describes the file and updates the internal DispRec that specifies its appearance in the pick list.

FormatMultiFile is useful when writing a general purpose file manager such as the DESKPOP Files window. Suppose that the user asks to change a marked set of files to read-only status and that the list is set up to show each file's attribute. Rather than re-reading the directory after making the changes, you can directly update each file's DirRec (using the pointer returned by GetMultiDirPtr) and then call FormatMultiFile to update its display string.

Example

```
{Scan all the files}
for N := 1 to Dir.GetMatchingFileCount do
  {Check for selected ones}
  if Dir.ItemIsSelected(N) then begin
    {Change the file to read-only}
    ...
    {Mark it read-only as far as OPDIR is concerned}
    with Dir.GetMultiDirPtr(N)^ do
      Attr := Attr or ReadOnly;
    {Update its display string}
    Dir.FormatMultiFile(N);
  end;
```

Implements the scenario described above. For each marked file, the attribute is set to read-only and OPDIR's list is updated to reflect this change. The next time the list is drawn (by calling Process, most likely), the changes will appear.

See Also

GetMultiDirPtr

GetFileName

Declaration

```
function GetFileName(Mask : String; FileAttr : Byte; var Name : PathStr) : Word;
```

Purpose

Set a new mask, pick a name, and return. Returns status.

Description

GetFileName is a shell around DirList's Process method. Calling GetFileName makes it easier to select a single filename for a program. The object that GetFileName references must have already been initialized by calling its constructor. Any customization (reject, sort, format options, etc.) must have already been set up. GetFileName works only with directory lists initialized to pick a single file at a time.

Mask is a search mask for the file to get. It's usually just the raw string a user entered in response to an "Enter a filename:" prompt. Prior to calling GetFileName, you should trim leading and trailing white space from the raw string; you may also want to add a default extension to it. The Mask string may name an existing file, a valid new file, a drive or directory, an expression with wild cards, or totally erroneous gibberish. GetFileName's job is to sort these out.

FileAttr is the DOS file attribute to use when selecting files. To avoid selecting read-only, system, and hidden files, specify 0 or Directory for FileAttr. If you set the Directory bit, the user will be able to see subdirectory entries in the pick list and, by selecting such entries, will be able to browse around an entire hard disk.

Name returns the selected file name, or an empty string if an error occurred.

GetFileName returns a status code: 0 for success, else an error code consistent with those described in Appendix D.

Using the Mask, GetFileName updates the DirList's internal data structures and checks to see how many matching files there are. If there are none, GetFileName exits with one of the following combinations of return values:

```
GetFileName = epWarning+ecFileNotFound, Name = Mask
```

Mask specifies a valid file name but the file doesn't exist

```
GetFileName = epWarning+ecNoMoreFiles, Name = ''
```

Mask has wildcards or specifies a directory, but there are no matching files in that location

```
GetFileName = epWarning+ecPathNotFound, Name = ''
```

Mask specifies an invalid drive or directory

It's also possible for GetFileName to return some other error, like out of memory.

If a) there is exactly one file that matches the Mask, b) the mask has no wildcards, c) the directory window is not already visible when GetFileName is called, and d) the diExitIfOne DirList option is enabled, then GetFileName doesn't display or process the pick list. Instead it immediately returns a status code of 0 and the complete pathname of the file in the Name parameter.

Otherwise, GetFileName calls Process to display a pick list of the matched files. The user can browse around and eventually exit the list. If the user makes a selection (exits with the command ccSelect), GetFileName returns a status code of 0 and the complete pathname of the file in the Name parameter. If the user exits with some other key (<Esc>, for example), GetFileName returns epWarning+ecNoPickSelection and Name = ''.

If the directory window wasn't visible at the time GetFileName was called, it is erased before exiting.

Because of the variety of errors that GetFileName can encounter, you should always check its return value before using the value of Name.

Example

```
uses
  Dos, OpString, OpCrt, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  S : PathStr;
  Status : Word;
begin
  {Initialize the DirList and select various options}
  Dir.InitCustom(25, 5, 55, 20, DefaultColorSet,
                 DefWindowOptions or wBordered,
                 MaxAvail, PickHorizontal, SingleFile);
  Dir.SetPadSize(1, 1);
  Dir.diOptionsOn(diOptimizeSize);
  Dir.AddMaskHeader(True, 1, 30, heTC);

  {Prompt for a mask and select a file}
  Write('File to pick: ');
  ReadLn(S);
  Status := Dir.GetFileName(Trim(S), Directory, S);
  if Status <> 0 then
    WriteLn('Status code: ', Status)
  else
    WriteLn('Selected: ', S);
end.
```

Displays a directory list and accepts one file selection.

See Also

ChangeDirectory
ExistFile (OPDOS)
Process

GetMatchingFileCount / GetMatchingSize

Declaration

```
function GetMatchingFileCount : Word;
```

Purpose

Return number of files matched in last directory.

Description

This is the same number of files that appeared in the pick list (or that will *eventually* appear in the pick list if the PreloadDirList method is called). It doesn't include any files rejected by the Reject function.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.PreLoadDirList;
WriteLn(Dir.GetMatchingFileCount,' files matched.');
```

Preloads a directory list and displays a count of the files that matched the mask.

See Also

FileCapacity

Declaration

```
function GetMatchingSize(ClusterSize : Word) : LongInt;
```

Purpose

Return number of bytes in all matched files.

Description

Uses the same rules that DOS does to round the exact file size up to a specified ClusterSize. If you want to know the exact sum of the file sizes, specify a ClusterSize of 1. Otherwise, use the DOS cluster size, which is typically 2048. You can find the actual cluster size for a DOS disk by using DOS.GetDiskInfo.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.PreLoadDirList;
WriteLn(Dir.GetMatchingSize(2048), ' bytes in the matching files.');
```

Reports the number of bytes used by all the files in the current directory.

Declaration
```
procedure GetMultiDirRec(N : Word; var D : DirRec);
```

Purpose
Return full DirRec for specified file.

Description
Accesses the internal array of DirRec to return the DirRec for the specified file. If N is zero or greater than the current number of files, the DirRec is filled with zeros.

Example
```
var
  N : Word;
  DR : DirRec;

{...Do multi-pick...}
Dir.InitSequence(N);
while Dir.HaveSelected(N) do begin
  Dir.GetMultiDirRec(N, DR);
  WriteLn('Selected: ', DR.Name, ' ', DR.Size);
  Dir.NextSelected(N);
end;
```

Retrieves the directory record of each selected entry and displays the file name and size.

See GetMultiFile for a complete program example for selecting multiple files.

See Also
GetMultiFile

Declaration
```
procedure GetMultiDirPtr(N : Word) : DirRecPtr;
```

Purpose
Return pointer to DirRec for specified file.

Description
Returns the address of the DirRec for the specified file. This pointer can be used to modify the DirRec. If N is zero or greater than the current number of files, GetMultiDirPtr returns nil. See FormatMultiFile for an application of this method.

See Also
FormatMultiFile

GetMultiFile / GetMultiPath

Declaration

```
function GetMultiFile(N : Word) : FileStr;
```

Purpose

Return 12-character name of specified file.

Description

The returned string is taken directly from the internal array of DirRec. It is always returned in uppercase. If N is zero or greater than the current number of files, the empty string is returned.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  N : Word;
begin
  Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, MultipleFile);
  Dir.SetMask('*.*', AnyFile);
  Dir.SetSelectMarker('<', '>');
  Dir.Process;
  Dir.Erase;
  Dir.InitSequence(N);
  while Dir.HaveSelected(N) do begin
    WriteLn('Selected: ', Dir.GetMultiFile(N));
    Dir.NextSelected(N);
  end;
  Dir.Done;
end.
```

Displays the names of all files selected during a multiple file selection.

See Also

GetMultiDirRec

Declaration

```
function GetMultiPath(N : Word) : PathStr;
```

Purpose

Return full pathname of specified file.

Description

Works like GetMultiFile but returns the complete pathname (including drive and directory) of the specified file.

See Also

GetMultiDirRec

Declaration
```
function GetSelectedDir : DirStr;
```

Purpose
Return directory of selected file.

Description
The directory string returned includes the drive and subdirectory. It is not terminated with a backslash (unless the directory is the root directory). GetSelectedDir returns the last directory whose DOS directory was read.

Example
```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.Process;
Dir.Erase;
if Dir.GetLastCommand = ccSelect then
  WriteLn('selected ', Dir.GetSelectedFile,
          ' in directory ', Dir.GetSelectedDir);
```

Displays the name and directory of the selected file.

See Also
GetSelectedFile GetSelectedPath

Declaration
```
procedure GetSelectedDirRec(var D : DirRec);
```

Purpose
Return full DirRec for selected file.

Description
The DirRec holds not only the file's name, but also its size, timestamp, and attribute. This method should not be called until the Process command exits without error. Otherwise the returned value will be filled with zeros.

Example
```
var
  DR : DirRec;
...
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.Process;
Dir.Erase;
if Dir.GetLastCommand = ccSelect then begin
  Dir.GetSelectedDirRec(DR);
  WriteLn('selected ', DR.Name, ' ', DR.Size, ' bytes');
end;
```

Displays the name and size of the selected file.

See Also
GetSelectedFile GetSelectedPath

GetSelectedFile / GetSelectedPath

Declaration

```
function GetSelectedFile : FileStr;
```

Purpose

Return 12-character name of selected file.

Description

The name is always returned in uppercase. It does not include the directory name of the file. This method should not be called until the Process command exits without error. Otherwise the returned value will be an empty string.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.Process;
Dir.Erase;
if Dir.GetLastCommand = ccSelect then
  WriteLn('selected ', Dir.GetSelectedFile);
```

Uses the GetSelectedFile function to display whatever file name is selected from the list.

See Also

GetSelectedDirRec

Declaration

```
function GetSelectedPath : PathStr;
```

Purpose

Return full pathname of selected file.

Description

This method combines GetSelectedDir and GetSelectedFile. The pathname includes drive, directory, and filename. This method should not be called until the Process command exits without error. Otherwise the returned value will be an empty string.

See Also

GetSelectedDir

Declaration

```
function GetSelectedSize(ClusterSize : Word) : LongInt;
```

Purpose

Return total size of all selected files.

Description

This method is meant to be used with multiple choice lists. It sums the size of each marked file after rounding it up to the nearest cluster. See GetMatchingSize for more information about ClusterSize.

To get the size of a single file selection, use GetSelectedDirRec.

Example

```
{...do multi-pick...}
WriteLn('Total size of selected files: ', Dir.GetSelectedSize(2048));
```

Displays the total size (in bytes) of all selected files. See GetMultiFile for a complete program example of multiple file selection.

See Also

GetMatchingSize

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte;
                 HeapToUse : Word;
                 PickOrientation : pkGenlProc;
                 CommandInit : diInitCommandProc);
```

Purpose

Initialize a directory list.

Description

This constructor instantiates a DirList using the default window options (DefWindowOptions), the default color set (DefaultColorSet), the default pick options (DefPickOptions), and the default directory options (DefDirOptions). See InitCustom for more details.

Example

```
if not Dir.Init(25, 5, 55, 20,          {Window coordinates}
                MaxAvail,               {Max heap space for files}
                PickHorizontal,         {Pick orientation}
                SingleFile)             {Command handler}
then begin
  WriteLn('Failed to Init DirList,  Status = ', InitStatus);
  Halt;
end;
...
```

Instantiates Dir with default window and color options.

See Also

InitCustom

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet;
                       Options : LongInt;
                       HeapToUse : Word;
                       PickOrientation : pkGenlProc;
                       CommandInit : diInitCommandProc);
```

Purpose

Initialize a directory list with custom window options and colors.

Description

X1, Y1, X2, and Y2 specify the active window coordinates within which the directory list will appear. Colors and Options are interpreted exactly the same as for a PickList. For the purposes of specifying colors, note that DirList treats files as pkNormal items and subdirectories as pkAlternate items.

HeapToUse is the number of bytes of heap space to allocate for filename storage. InitCustom will allocate at most $FFF1 bytes. If it cannot allocate the lesser of $FFF1 or HeapToUse bytes, the constructor will fail. See the FileCapacity method for information about the relationship between the heap space and the maximum number of files managed.

PickOrientation specifies the orientation of the underlying PickList. It can take the value PickVertical, PickHorizontal, or PickSnaking.

CommandInit specifies whether the directory list manages multiple file selections or just a single one. Acceptable values for this parameter are MultipleFile or SingleFile. These values are actually procedures interfaced by OPDIR.

The PickList InitCustom constructor performs most of the work. Beyond that, DirList.InitCustom sets up the default filename rejection filter (RejectNone), the default sort routine (SortNone), and the default display format (SetNameFormat). See the introduction to this section for more information about these options.

The constructor allocates heap space for filename storage and--if the MultipleFile command handler is specified--for the selection BitSet. Heap space for the BitSet is allocated in addition to the specified HeapToUse parameter. (It is typically a small amount, however, since the BitSet packs 8 files into each byte. Thus, 1600 files are managed with a 200 byte BitSet.)

The PickCommands command processor is assigned to the DirList. The object is assigned a unit code of ucDir (for the purposes of identifying it to error and help handlers). See the discussion of "Directory Commands" in the introduction for more background.

Note that the DirList is assigned the default directory options from DefDirOptions. You can change these options by calling diOptionsOn or diOptionsOff after the constructor returns.

If the constructor fails, check the InitStatus variable for a detailed error code.

Example

```
if not Dir.InitCustom(25, 5, 55, 20,                {Window coordinates}
                      DefaultColorSet,              {ColorSet}
                      DefWindowOptions or wBordered, {Window options}
                      MaxAvail,                     {Max heap space for files}
                      PickHorizontal,               {Pick orientation}
                      SingleFile)                   {Command handler}
then begin
  WriteLn('Failed to Init DirList,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates Dir with a bordered window and default colors.

See Also

FileCapacity PickList.InitDeluxe

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a list from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to DirList.Store. Load first calls PickList.Load to read the underlying PickList. Then it reads data specific to the DirList, including formatting masks, the reject set, and procedure pointers to reject, sort, and format routines.

The stream registration procedure for a DirList is DirListStream. There are also a number of procedure pointers that must be registered independently. The example program below demonstrates how. See Store for more details.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  S : BufIdStream;
begin
  {Load directory window from stream}
  S.Init('DIRLIST.STM', SOpen, 1024);
  S.RegisterHier(DirListStream);
  S.RegisterPointer(ptPickHorizontal, @PickHorizontal);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.RegisterPointer(ptDirAllFormat, @AllFormat);
  S.RegisterPointer(ptDirSortName, @SortName);
  S.Get(Dir);
  S.Done;

  {Give it a try}
  Dir.Process;
  Dir.Done;
end.
```

Instantiates Dir from the stream created by the Store example. Note that we specifically register the pick orientation routine, the pick command handler, the directory formatting function, and the directory sorting procedure.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

PickList.Load Store

PreloadDirList / Process

Declaration

```
procedure PreloadDirList;
```

Purpose

Read directory in advance.

Description

This method may be called before the directory window is drawn or a selection is requested via Process. PreloadDirList uses the selection mask specified with SetMask to read the DOS directory and then calls the user-defined routines to reject, sort, and format the files. A system of internal flags avoids repetition of any part of this work until some external change requires it.

PreloadDirList may be used to avoid displaying the pick window when no suitable files are found. PreloadDirList also is useful to localize the potentially significant delay (up to several seconds for large directories with complex date/time formatting) which would otherwise occur when the Process method is called.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.PreLoadDirList;
Result := Dir.GetLastError;
if Result <> 0 then
  WriteLn('Fatal error ', Result)
else
  Dir.Process;
Dir.Done;
```

Load the directory list before calling Process.

See Also

GetFileName

Declaration

```
procedure Process; virtual;
```

Purpose

Process pick commands leading to a file choice.

Description

The DirList Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It calls PickList.Process to perform command processing. It exits on any command that is a PickList exit command, with the possible exception of ccSelect. When ccSelect is executed and the current item is a subdirectory, DirList.Process does not exit; instead, it reads the files in the specified subdirectory and continues processing.

Process calls several user-defined routines. In addition to the routines applicable to the PickList, a DirList calls routines for filename filtering, directory sorting, and filename formatting. See the introduction to this section for more information about these routines.

Example

For a complete example, see the introduction to this section.

See Also

ChangeDirectory
CommandWindow.Process
GetFileName
PickList.Process

Declaration

```
function Reject(var X : DirRec) : Boolean; virtual;
```

Purpose

Return True to exclude a particular file from list.

Description

OPDIR calls this virtual method to determine whether each raw file record (X) should be added to the list of files to display. If Reject returns True, the filename will be ignored and the directory listing will make no mention of the file. DirList's implementation of Reject simply calls another user-defined routine via a procedure pointer.

See the discussion of "Filename Filtering" in the introduction to this section for more information.

See Also

SetRejectFunc

Declaration

```
procedure RemoveMultiFile(N : Word);
```

Purpose

Remove specified file from list.

Description

RemoveMultiFile removes the specified file from all of the DirList's records; at the next call to Draw or Process the file will no longer appear. The call to RemoveMultiFile shifts the items above N down by one in the list. If N specifies the currently highlighted file, the highlight will move to the file following N, unless N is the last file.

This method is useful when writing a full-featured file manager like the one in DESKPOP, where you may have a command to physically delete marked files. If you call RemoveMultiFile to remove each such file then you won't need to force DirList to read, sort, and format the list again. Be careful not to call Process again after removing *all* files, however.

Example

```
repeat
  {...do multi-pick...}
  Dir.InitSequence(N);
  while Dir.HaveSelected(N) do begin
    Dir.RemoveMultiFile(N);
    Dec(N);
    Dir.NextSelected(N);
  end;
until Finished;
```

Removes all selected files from the directory list, then repeats the pick. We decrement the value of N to avoid skipping over selected files. Note that the example doesn't physically erase any files; it just removes them from the directory display.

See RemoveSelectedFiles for an easier way to do this. See GetMultiFile for a complete example of multiple file selections.

See Also

RemoveSelectedFile RemoveSelectedFiles

RemoveSelectedFile / RemoveSelectedFiles

Declaration

```
procedure RemoveSelectedFile;
```

Purpose

Remove current file from pick list.

Description

This method calls RemoveMultiFile with N set to the current file index.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Finished := False;
repeat
  Dir.Process;
  case Dir.GetLastCommand of
    ccSelect : Dir.RemoveSelectedFile;
    ccQuit   : Finished := True;
  end;
until Finished;
```

Removes each selected file entry from the list.

See Also

RemoveMultiFile

Declaration

```
procedure RemoveSelectedFiles;
```

Purpose

Remove all selected files from list.

Description

This method is intended to be used with multi-file pick lists. It removes all marked files from the list by repeatedly calling RemoveMultiFile.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, MultipleFile);
Dir.SetMask('*.*', AnyFile);
Dir.SetSelectMarker('<', '>');
Finished := False;
repeat
  Dir.Process;
  if Dir.GetSelectedCount = 0 then
    Finished := True
  else
    Dir.RemoveSelectedFiles;
until Finished;
```

Removes all selected files after a multiple file selection.

See Also

RemoveMultiFile

Declaration

```
procedure ResetList;
```

Purpose

Force reread of directory when it may have changed.

Description

This method sets a number of internal DirList flags which indicate that the Draw, PreloadDirList, and Process methods must again read the DOS directory, sort the files, format all filename strings, optionally resize the pick window, reposition the highlight bar, and update the mask header. DirList normally resets the list in this way only when you change the search mask (via SetMask). You may need to reset the list explicitly if your program adds or deletes files, or otherwise updates the directory, without changing the DirList search mask.

DirList provides other methods to remove files from the directory list (see RemoveMultiFile), but it provides none to add files. The RemoveMultiFile approach will be faster than ResetList when it is practical and appropriate.

Example

```
var
  F : File;
begin
  {...Init/setup Dir...}
  Finished := False;
  repeat
    Dir.Process;
    case Dir.GetLastCommand of
      ccSelect :
        begin
          Assign(F, Dir.GetSelectedFile);
          System.Erase(F);
          ResetList;
        end;
      ccQuit :
        Finished := True;
    end;
  until Finished;
end.
```

Selecting a file causes that file to be deleted; ResetList assures that the directory is reread when Process is called again.

See Also

RemoveMultiFile SetMask

SelectedFileInCurDir / SelectItem

Declaration

```
function SelectedFileInCurDir : Boolean;
```

Purpose

Return true if selected file is in current directory.

Description

SelectedFileInCurDir compares the string returned by GetSelectedDir to that returned by Turbo Pascal's GetDir procedure with a drive number of 0. Therefore it returns True if the last file selected in Process is in the current directory of the current drive.

Example

```
{...Pick a file...}
if Dir.SelectedFileInCurDir then
  Fname := Dir.GetSelectedFile
else
  Fname := Dir.GetSelectedPath;
```

If the selected file is in the current directory, it is referred to just by its filename; otherwise, the full pathname is used.

See Also

GetSelectedDir GetSelectedPath

Declaration

```
procedure SelectItem(N : Word); virtual;
```

Purpose

Select an item.

Description

DirList.SelectItem overrides the method of the same name in PickList in order to prevent marking of subdirectories. A call to DirList.SelectItem where N refers to a subdirectory item is simply ignored.

This version of SelectItem makes the DirList behave as described in the "Directory Commands" introduction. However, it means that you can't use a DirList to mark multiple subdirectories for some common operation. If you need to do so, use a PathList object instead, or else derive a new type from DirList that re-overrides the SelectItem method.

Example

```
{...set up for multi-pick...}
repeat
  Dir.Process;
  case Dir.GetLastCommand of
    ccUser0 : for I := 1 to diCurFiles do
                Dir.SelectItem(I);
    ccUser1 : for I := 1 to diCurFiles do
                Dir.DeSelectItem(I);
    ccQuit : Finished := True;
  end;
until Finished;
```

Uses an exit command (ccUser0) to select all displayed files with one keystroke. Subdirectories will not be marked.

See Also

PickList.SelectItem

Declaration
```
procedure SetDirAttr(Color, Mono : Byte);
```

Purpose
Set video attributes for directories.

Description
SetDirAttr simply makes a call to SetPickAttr for the pkAlternate item category in its unselected state. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example
```
Dir.SetDirAttr($02, $70);
```
Displays directory names in green on black for color monitors, reverse video for monochrome monitors.

See Also
PickList.SetPickAttr SetSelectDirAttr

Declaration
```
procedure SetFileAttr(Color, Mono : Byte);
```

Purpose
Set video attributes for files.

Description
SetFileAttr simply makes a call to SetPickAttr for the pkNormal item category in its unselected state. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example
```
Dir.SetFileAttr($06, $70);
```
Displays file names in brown on black for color monitors, reverse video for monochrome monitors.

See Also
PickList.SetPickAttr SetSelectFileAttr

SetMask

Declaration

```
procedure SetMask(Mask : String; FileAttr : Byte);
```

Purpose

Set search mask and attribute and reset list.

Description

SetMask stores a new search mask and search attribute in data fields of the object, then calls ResetList. The next time PreloadDirList, Draw, or Process is called, the DirList will read the DOS directory, sort the list, format the files, and so on.

Mask should be a valid DOS pathname or wildcard combination. (Invalid masks may be specified as well; the DirList will return a non-fatal error after a later call to PreloadDirList, Draw, or Process.) The mask need not specify the current drive or directory. An empty mask is interpreted the same as '*.*'.

FileAttr is a DOS file attribute to match. The DOS unit defines constants that specify the attributes. Normal files have attribute 0; you can add the following constants to specify other attributes.

```
ReadOnly  = $01;
Hidden    = $02;
SysFile   = $04;
VolumeID  = $08;
Directory = $10;
Archive   = $20;
AnyFile   = $3F;
```

Note that DOS uses additive rules in applying these attributes. For example, if you specify FileAttr = ReadOnly, you get read-only files in addition to normal files. If you specify Directory, you get directories and non-directories. You get Archive files whether or not you specify the Archive bit. You can use the DirList's reject function to filter files more precisely.

Examples

```
Dir.SetMask('*.*', AnyFile);
```

Lists all files in the current directory, including subdirectories and special files like System and Hidden files.

```
Dir.SetMask('*.PAS', 0);
```

Lists all files in the current directory that have a .PAS extension; no subdirectories and no special files are included.

See Also

ResetList

Declaration

```
procedure AddMaskHeader(FullPath : Boolean; PadSize, MaxLen : Byte;
                        Posn : HeaderPosType);
```

Purpose

Add a specialized header to display the search mask.

Description

AddMaskHeader makes a call to wFrame.AddHeader after it initializes internal data structures to keep track of the mask header. DirList will then automatically update it to display the current search mask whenever it is changed.

If FullPath is True, the header will show the drive and directory as well as the search mask itself.

PadSize specifies the number of spaces to add to each end of the header before displaying it. MaxLen is the total number of characters to reserve for the header, including any padding. If the actual mask would exceed this length, DirList will remove internal characters from the header before displaying it. In order to provide a meaningful display, AddMaskHeader always makes sure that MaxLen-2*PadSize is at least six characters.

Posn specifies the frame-relative position of the header (heTL, heTC, heTR, heBL, heBC, or heBR).

Whenever DirList updates the mask header, it sets a flag in the internal DirList options word. A user-defined move procedure (see PickList.PreMove) can take advantage of this flag to perform a special action--updating a file and byte count status line, for example--whenever a new directory appears. It is the user routine's responsibility to reset the flag. Here's a sequence of code that would appear in the move procedure to take advantage of this flag:

```
if DirListPtr(P)^.diOptionsAreOn(diUserNewHdr) then begin
  {User is viewing a different directory}
  ...
  {Reset the flag}
  diOptionsOff(diUserNewHdr);
end;
```

Example

```
Dir.SetMask('*.*', AnyFile);
Dir.AddMaskHeader(True, 1, 20, heTC);
```

If the current directory were 'C:\TEST', then the directory window would display the header ' C:\TEST*.* ' centered in the top border. If the user navigated to another directory, DirList would update the header accordingly. If the user got to a directory named C:\TEST\LONGDIR\NEXTDIR, then DirList would shorten the header to read ' C:...R\NEXTDIR*.* '. Note that the length of the header is 20 characters.

See Also

SetMask

AddRejectString

Declaration

```
procedure AddRejectString(S : String);
```

Purpose

Add a string to the set to reject.

Description

DirList manages an internal StringSet object. You can use this StringSet to reject files based on their extensions or perhaps other characteristics. In combination with the RejectExtensions reject function, the internal StringSet is interpreted specifically as a set of extensions. In this case, the parameter S should be a particular extension ('BAK', for example) that you don't want to appear in the directory list.

Because AddRejectString must allocate heap space to store the specified string and supporting data structures, it's possible for this routine to fail with the error epNonFatal+ecOutOfMemory. Call GetLastError to check the status.

For more information, see the discussion of "Filename Filtering" at the introduction to this section.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.AddRejectString('BAK');
if Dir.GetLastError = 0 then
  Dir.SetRejectFunc(RejectExtensions)
else
  {Out of memory} ;
Dir.Process;
```

Reject all files with an extension of 'BAK'.

See Also

ClearRejectStrings StringSet.Add

Declaration

```
procedure ChangeDirectory(Mask : String; ChangeCode : Byte);
```

Purpose

Change to specified directory, or let user browse around to do so.

Description

ChangeDirectory is a shell around DirList's Process method. It is provided to make it easier to use the DirList as a way to change directories. The object that ChangeDirectory references must have already been initialized by calling its constructor. Any customization (reject, sort, format options, etc.) must have already been set up. ChangeDirectory works only with directory lists initialized to pick a single file at a time.

Mask is the name of the directory to change to. If it is a non-empty string and contains no wildcards, ChangeDirectory will attempt to change directly to that directory by calling Turbo Pascal's ChDir procedure, without displaying a pick list at all. If ChDir fails, ChangeDirectory reports the warning epWarning+ecPathNotFound, which you can check by calling GetLastError.

If mask is an empty string or contains wildcards, ChangeDirectory attempts to build a pick list of the matching directories. It does so by temporarily switching to the RejectFiles reject function, which forces DirList to display only subdirectories in its pick list. It converts the empty string to a '*.*' search mask. If no matching directories are found, ChangeDirectory again reports the warning epWarning+ecPathNotFound and exits.

If matching directories are found, ChangeDirectory calls Process to display a pick list of these items. If the user scrolls around this list and presses <Enter> on a certain directory, DirList will use the search mask again to find matching directories in that location.

Here's where the ChangeCode parameter comes in. ChangeCode is the command code for a particular DirList exit command. When the user presses a keystroke combination that maps to this exit command, Process will exit and ChangeDirectory will change the current directory to the last one listed by the pick list. If the user exits with any command besides ChangeCode, ChangeDirectory doesn't change the current directory.

If the DirList window wasn't visible when ChangeDirectory was called, it is erased again before ChangeDirectory exits.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
begin
  if not Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal) then
    Halt;
  Dir.ChangeDirectory('*.*', ccQuit);
end.
```

Display a list of directories and switch to the selected directory when the user presses <Esc> (ccQuit). Pressing <Esc> to change directories is somewhat counter-intuitive. You might either want to assign a different exit key or evaluate PathList.GetPathName for an alternative method of switching directories.

See Also

PathList.GetPathName

ClearRejectStrings / diOptionsAreOn

Declaration
```
procedure ClearRejectStrings;
```

Purpose
Clear the set of strings to reject.

Description
DirList's internal StringSet is cleared and disposed. The entire directory will be reread the next time Process or PreloadDirList is called, in order to avoid missing any files that are no longer rejected.

Example
```
Dir.ClearRejectStrings;
```
Clear any previously defined reject strings.

See Also
AddRejectString StringSet.Done

Declaration
```
function diOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This function tests the directory list options, which are described in the Constant declarations near the beginning of this section.

Example
```
if Dir.diOptionsAreOn(diDirsUpcase) then
  {take some action};
```
Takes action if the diDirUpcase option is on.

See Also
diOptionsOff diOptionsOn

Declaration
```
procedure diOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadDirOptions (see the Constants section, above).

Example
```
Dir.diOptionsOff(diOptimizeSize+diDirsUpcase+diFilesUpcase);
```
Turns off three options at once: window size will not be optimized, directory and file names will be forced to lower case.

Declaration
```
procedure diOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadDirOptions (see the Constants section, above).

Example
```
Dir.diOptionsOn(diOptimizeSize+diDirsUpcase+diFilesUpcase);
```
Turns on three options at once: optimize window size, display both directory and file names in upper case.

Done / FileCapacity

Declaration

```
destructor Done; virtual;
```

Purpose

Dispose of directory list.

Description

This destructor deallocates the heap space that DirList uses to store directory arrays, disposes of its internal StringSet (if any), and then calls PickList.Done.

Example

```
Dir.Done;
```

Disposes of the heap space used by Dir.

See Also

PickList.Done StringSet.Done

Declaration

```
function FileCapacity : Word;
```

Purpose

Return the maximum number of files the list can hold.

Description

When a DirList is initialized, you specify how many bytes of heap space it can use for file storage. This heap space is split between two arrays: an array of DirRec (22 bytes) and an array of DispRec (2 bytes plus the maximum length of each formatted file string). FileCapacity divides the heap space by the bytes per file, and returns the maximum number of files that the DirList can manage.

When a DirList attempts to read a directory holding more files than it can handle, it generates a warning (epWarning+ecTooManyFiles), but will continue processing with the files that it could store.

Example

```
if not Dir.InitCustom(25, 5, 55, 20,                {Window coordinates}
                      DefaultColorSet,              {ColorSet}
                      DefWindowOptions or wBordered, {Window options}
                      MaxAvail,                     {Heap space for files}
                      PickVertical,                 {Pick orientation}
                      SingleFile)                   {Command handler}
then begin
  WriteLn('Failed to Init DirList,  Status = ', InitStatus);
  Halt;
end;
...
Dir.SetNameSizeTimeFormat('<dir>', 'Mm/dd/yy', 'Hh:mmt');
WriteLn('Maximum files: ', Dir.FileCapacity);
```

Assuming that at least 64K of contiguous heap space is available, this example will indicate that 813 can be managed by the DirList. If a simpler display format is used, Dir will be able to store correspondingly more files. For example, using the shorter format activated by SetNameFormat, the list can manage 1477 files.

See Also

Init

Declaration
```
procedure Format(var X : DirRec; var pkCat : Byte; var S : String); virtual;
```

Purpose
Return a formatted string and category for a directory entry.

Description
OPDIR calls this virtual method to convert each raw file record (X) into a formatted string (S) and pick list category (pkCat). The method may be overridden by derived types to implement a specific formatting style. DirList's implementation of Format simply passes on the call to another user-defined format procedure.

See the discussion of "Filename Formatting" near the start of this section for more information and for complete examples.

See Also
SetXxxFormat

Declaration
```
procedure FormatMultiFile(N : Word);
```

Purpose
Rebuild the format string for specified file.

Description
The method calls the format function for the N'th file. It reads the internal DirRec that describes the file and updates the internal DispRec that specifies its appearance in the pick list.

FormatMultiFile is useful when writing a general purpose file manager such as the DESKPOP Files window. Suppose that the user asks to change a marked set of files to read-only status and that the list is set up to show each file's attribute. Rather than re-reading the directory after making the changes, you can directly update each file's DirRec (using the pointer returned by GetMultiDirPtr) and then call FormatMultiFile to update its display string.

Example
```
{Scan all the files}
for N := 1 to Dir.GetMatchingFileCount do
  {Check for selected ones}
  if Dir.ItemIsSelected(N) then begin
    {Change the file to read-only}
    ...
    {Mark it read-only as far as OPDIR is concerned}
    with Dir.GetMultiDirPtr(N)^ do
      Attr := Attr or ReadOnly;
    {Update its display string}
    Dir.FormatMultiFile(N);
  end;
```

Implements the scenario described above. For each marked file, the attribute is set to read-only and OPDIR's list is updated to reflect this change. The next time the list is drawn (by calling Process, most likely), the changes will appear.

See Also
GetMultiDirPtr

GetFileName

Declaration

```
function GetFileName(Mask : String; FileAttr : Byte; var Name : PathStr) : Word;
```

Purpose

Set a new mask, pick a name, and return. Returns status.

Description

GetFileName is a shell around DirList's Process method. Calling GetFileName makes it easier to select a single filename for a program. The object that GetFileName references must have already been initialized by calling its constructor. Any customization (reject, sort, format options, etc.) must have already been set up. GetFileName works only with directory lists initialized to pick a single file at a time.

Mask is a search mask for the file to get. It's usually just the raw string a user entered in response to an "Enter a filename:" prompt. Prior to calling GetFileName, you should trim leading and trailing white space from the raw string; you may also want to add a default extension to it. The Mask string may name an existing file, a valid new file, a drive or directory, an expression with wild cards, or totally erroneous gibberish. GetFileName's job is to sort these out.

FileAttr is the DOS file attribute to use when selecting files. To avoid selecting read-only, system, and hidden files, specify 0 or Directory for FileAttr. If you set the Directory bit, the user will be able to see subdirectory entries in the pick list and, by selecting such entries, will be able to browse around an entire hard disk.

Name returns the selected file name, or an empty string if an error occurred.

GetFileName returns a status code: 0 for success, else an error code consistent with those described in Appendix D.

Using the Mask, GetFileName updates the DirList's internal data structures and checks to see how many matching files there are. If there are none, GetFileName exits with one of the following combinations of return values:

```
GetFileName = epWarning+ecFileNotFound, Name = Mask
```

Mask specifies a valid file name but the file doesn't exist

```
GetFileName = epWarning+ecNoMoreFiles, Name = ''
```

Mask has wildcards or specifies a directory, but there are no matching files in that location

```
GetFileName = epWarning+ecPathNotFound, Name = ''
```

Mask specifies an invalid drive or directory

It's also possible for GetFileName to return some other error, like out of memory.

If a) there is exactly one file that matches the Mask, b) the mask has no wildcards, c) the directory window is not already visible when GetFileName is called, and d) the diExitIfOne DirList option is enabled, then GetFileName doesn't display or process the pick list. Instead it immediately returns a status code of 0 and the complete pathname of the file in the Name parameter.

Otherwise, GetFileName calls Process to display a pick list of the matched files. The user can browse around and eventually exit the list. If the user makes a selection (exits with the command ccSelect), GetFileName returns a status code of 0 and the complete pathname of the file in the Name parameter. If the user exits with some other key (<Esc>, for example), GetFileName returns epWarning+ecNoPickSelection and Name = ''.

If the directory window wasn't visible at the time GetFileName was called, it is erased before exiting.

Because of the variety of errors that GetFileName can encounter, you should always check its return value before using the value of Name.

Example

```
uses
  Dos, OpString, OpCrt, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  S : PathStr;
  Status : Word;
begin
  {Initialize the DirList and select various options}
  Dir.InitCustom(25, 5, 55, 20, DefaultColorSet,
                 DefWindowOptions or wBordered,
                 MaxAvail, PickHorizontal, SingleFile);
  Dir.SetPadSize(1, 1);
  Dir.diOptionsOn(diOptimizeSize);
  Dir.AddMaskHeader(True, 1, 30, heTC);

  {Prompt for a mask and select a file}
  Write('File to pick: ');
  ReadLn(S);
  Status := Dir.GetFileName(Trim(S), Directory, S);
  if Status <> 0 then
    WriteLn('Status code: ', Status)
  else
    WriteLn('Selected: ', S);
end.
```

Displays a directory list and accepts one file selection.

See Also

ChangeDirectory
ExistFile (OPDOS)
Process

GetMatchingFileCount / GetMatchingSize

Declaration

```
function GetMatchingFileCount : Word;
```

Purpose

Return number of files matched in last directory.

Description

This is the same number of files that appeared in the pick list (or that will *eventually* appear in the pick list if the PreloadDirList method is called). It doesn't include any files rejected by the Reject function.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.PreLoadDirList;
WriteLn(Dir.GetMatchingFileCount,' files matched.');
```

Preloads a directory list and displays a count of the files that matched the mask.

See Also

FileCapacity

Declaration

```
function GetMatchingSize(ClusterSize : Word) : LongInt;
```

Purpose

Return number of bytes in all matched files.

Description

Uses the same rules that DOS does to round the exact file size up to a specified ClusterSize. If you want to know the exact sum of the file sizes, specify a ClusterSize of 1. Otherwise, use the DOS cluster size, which is typically 2048. You can find the actual cluster size for a DOS disk by using DOS.GetDiskInfo.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.PreLoadDirList;
WriteLn(Dir.GetMatchingSize(2048), ' bytes in the matching files.');
```

Reports the number of bytes used by all the files in the current directory.

Declaration
```
procedure GetMultiDirRec(N : Word; var D : DirRec);
```

Purpose
Return full DirRec for specified file.

Description
Accesses the internal array of DirRec to return the DirRec for the specified file. If N is zero or greater than the current number of files, the DirRec is filled with zeros.

Example
```
var
  N : Word;
  DR : DirRec;

{...Do multi-pick...}
Dir.InitSequence(N);
while Dir.HaveSelected(N) do begin
  Dir.GetMultiDirRec(N, DR);
  WriteLn('Selected: ', DR.Name, ' ', DR.Size);
  Dir.NextSelected(N);
end;
```

Retrieves the directory record of each selected entry and displays the file name and size.

See GetMultiFile for a complete program example for selecting multiple files.

See Also
GetMultiFile

Declaration
```
procedure GetMultiDirPtr(N : Word) : DirRecPtr;
```

Purpose
Return pointer to DirRec for specified file.

Description
Returns the address of the DirRec for the specified file. This pointer can be used to modify the DirRec. If N is zero or greater than the current number of files, GetMultiDirPtr returns nil. See FormatMultiFile for an application of this method.

See Also
FormatMultiFile

GetMultiFile / GetMultiPath

Declaration

```
function GetMultiFile(N : Word) : FileStr;
```

Purpose

Return 12-character name of specified file.

Description

The returned string is taken directly from the internal array of DirRec. It is always returned in uppercase. If N is zero or greater than the current number of files, the empty string is returned.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  N : Word;
begin
  Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, MultipleFile);
  Dir.SetMask('*.*', AnyFile);
  Dir.SetSelectMarker('<', '>');
  Dir.Process;
  Dir.Erase;
  Dir.InitSequence(N);
  while Dir.HaveSelected(N) do begin
    WriteLn('Selected: ', Dir.GetMultiFile(N));
    Dir.NextSelected(N);
  end;
  Dir.Done;
end.
```

Displays the names of all files selected during a multiple file selection.

See Also

GetMultiDirRec

Declaration

```
function GetMultiPath(N : Word) : PathStr;
```

Purpose

Return full pathname of specified file.

Description

Works like GetMultiFile but returns the complete pathname (including drive and directory) of the specified file.

See Also

GetMultiDirRec

Declaration

```
function GetSelectedDir : DirStr;
```

Purpose

Return directory of selected file.

Description

The directory string returned includes the drive and subdirectory. It is not terminated with a backslash (unless the directory is the root directory). GetSelectedDir returns the last directory whose DOS directory was read.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.Process;
Dir.Erase;
if Dir.GetLastCommand = ccSelect then
  WriteLn('selected ', Dir.GetSelectedFile,
          ' in directory ', Dir.GetSelectedDir);
```

Displays the name and directory of the selected file.

See Also

GetSelectedFile GetSelectedPath

Declaration

```
procedure GetSelectedDirRec(var D : DirRec);
```

Purpose

Return full DirRec for selected file.

Description

The DirRec holds not only the file's name, but also its size, timestamp, and attribute. This method should not be called until the Process command exits without error. Otherwise the returned value will be filled with zeros.

Example

```
var
  DR : DirRec;
...
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.Process;
Dir.Erase;
if Dir.GetLastCommand = ccSelect then begin
  Dir.GetSelectedDirRec(DR);
  WriteLn('selected ', DR.Name, ' ', DR.Size, ' bytes');
end;
```

Displays the name and size of the selected file.

See Also

GetSelectedFile GetSelectedPath

GetSelectedFile / GetSelectedPath

Declaration
```
function GetSelectedFile : FileStr;
```

Purpose
Return 12-character name of selected file.

Description
The name is always returned in uppercase. It does not include the directory name of the file. This method should not be called until the Process command exits without error. Otherwise the returned value will be an empty string.

Example
```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.Process;
Dir.Erase;
if Dir.GetLastCommand = ccSelect then
  WriteLn('selected ', Dir.GetSelectedFile);
```
Uses the GetSelectedFile function to display whatever file name is selected from the list.

See Also
GetSelectedDirRec

Declaration
```
function GetSelectedPath : PathStr;
```

Purpose
Return full pathname of selected file.

Description
This method combines GetSelectedDir and GetSelectedFile. The pathname includes drive, directory, and filename. This method should not be called until the Process command exits without error. Otherwise the returned value will be an empty string.

See Also
GetSelectedDir

Declaration

```
function GetSelectedSize(ClusterSize : Word) : LongInt;
```

Purpose

Return total size of all selected files.

Description

This method is meant to be used with multiple choice lists. It sums the size of each marked file after rounding it up to the nearest cluster. See GetMatchingSize for more information about ClusterSize.

To get the size of a single file selection, use GetSelectedDirRec.

Example

```
{...do multi-pick...}
WriteLn('Total size of selected files: ', Dir.GetSelectedSize(2048));
```

Displays the total size (in bytes) of all selected files. See GetMultiFile for a complete program example of multiple file selection.

See Also

GetMatchingSize

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte;
                 HeapToUse : Word;
                 PickOrientation : pkGenlProc;
                 CommandInit : diInitCommandProc);
```

Purpose

Initialize a directory list.

Description

This constructor instantiates a DirList using the default window options (DefWindowOptions), the default color set (DefaultColorSet), the default pick options (DefPickOptions), and the default directory options (DefDirOptions). See InitCustom for more details.

Example

```
if not Dir.Init(25, 5, 55, 20,          {Window coordinates}
                MaxAvail,               {Max heap space for files}
                PickHorizontal,         {Pick orientation}
                SingleFile)             {Command handler}
then begin
  WriteLn('Failed to Init DirList,  Status = ', InitStatus);
  Halt;
end;
...
```

Instantiates Dir with default window and color options.

See Also

InitCustom

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet;
                       Options : LongInt;
                       HeapToUse : Word;
                       PickOrientation : pkGenlProc;
                       CommandInit : diInitCommandProc);
```

Purpose

Initialize a directory list with custom window options and colors.

Description

X1, Y1, X2, and Y2 specify the active window coordinates within which the directory list will appear. Colors and Options are interpreted exactly the same as for a PickList. For the purposes of specifying colors, note that DirList treats files as pkNormal items and subdirectories as pkAlternate items.

HeapToUse is the number of bytes of heap space to allocate for filename storage. InitCustom will allocate at most $FFF1 bytes. If it cannot allocate the lesser of $FFF1 or HeapToUse bytes, the constructor will fail. See the FileCapacity method for information about the relationship between the heap space and the maximum number of files managed.

PickOrientation specifies the orientation of the underlying PickList. It can take the value PickVertical, PickHorizontal, or PickSnaking.

CommandInit specifies whether the directory list manages multiple file selections or just a single one. Acceptable values for this parameter are MultipleFile or SingleFile. These values are actually procedures interfaced by OPDIR.

The PickList InitCustom constructor performs most of the work. Beyond that, DirList.InitCustom sets up the default filename rejection filter (RejectNone), the default sort routine (SortNone), and the default display format (SetNameFormat). See the introduction to this section for more information about these options.

The constructor allocates heap space for filename storage and--if the MultipleFile command handler is specified--for the selection BitSet. Heap space for the BitSet is allocated in addition to the specified HeapToUse parameter. (It is typically a small amount, however, since the BitSet packs 8 files into each byte. Thus, 1600 files are managed with a 200 byte BitSet.)

The PickCommands command processor is assigned to the DirList. The object is assigned a unit code of ucDir (for the purposes of identifying it to error and help handlers). See the discussion of "Directory Commands" in the introduction for more background.

Note that the DirList is assigned the default directory options from DefDirOptions. You can change these options by calling diOptionsOn or diOptionsOff after the constructor returns.

If the constructor fails, check the InitStatus variable for a detailed error code.

Example

```
if not Dir.InitCustom(25, 5, 55, 20,                {Window coordinates}
                      DefaultColorSet,              {ColorSet}
                      DefWindowOptions or wBordered, {Window options}
                      MaxAvail,                     {Max heap space for files}
                      PickHorizontal,               {Pick orientation}
                      SingleFile)                   {Command handler}
then begin
  WriteLn('Failed to Init DirList,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates Dir with a bordered window and default colors.

See Also

FileCapacity PickList.InitDeluxe

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a list from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to DirList.Store. Load first calls PickList.Load to read the underlying PickList. Then it reads data specific to the DirList, including formatting masks, the reject set, and procedure pointers to reject, sort, and format routines.

The stream registration procedure for a DirList is DirListStream. There are also a number of procedure pointers that must be registered independently. The example program below demonstrates how. See Store for more details.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Dir : DirList;
  S : BufIdStream;
begin
  {Load directory window from stream}
  S.Init('DIRLIST.STM', SOpen, 1024);
  S.RegisterHier(DirListStream);
  S.RegisterPointer(ptPickHorizontal, @PickHorizontal);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.RegisterPointer(ptDirAllFormat, @AllFormat);
  S.RegisterPointer(ptDirSortName, @SortName);
  S.Get(Dir);
  S.Done;

  {Give it a try}
  Dir.Process;
  Dir.Done;
end.
```

Instantiates Dir from the stream created by the Store example. Note that we specifically register the pick orientation routine, the pick command handler, the directory formatting function, and the directory sorting procedure.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

PickList.Load Store

PreloadDirList / Process

Declaration

```
procedure PreloadDirList;
```

Purpose

Read directory in advance.

Description

This method may be called before the directory window is drawn or a selection is requested via Process. PreloadDirList uses the selection mask specified with SetMask to read the DOS directory and then calls the user-defined routines to reject, sort, and format the files. A system of internal flags avoids repetition of any part of this work until some external change requires it.

PreloadDirList may be used to avoid displaying the pick window when no suitable files are found. PreloadDirList also is useful to localize the potentially significant delay (up to several seconds for large directories with complex date/time formatting) which would otherwise occur when the Process method is called.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Dir.PreLoadDirList;
Result := Dir.GetLastError;
if Result <> 0 then
  WriteLn('Fatal error ', Result)
else
  Dir.Process;
Dir.Done;
```

Load the directory list before calling Process.

See Also

GetFileName

Declaration

```
procedure Process; virtual;
```

Purpose

Process pick commands leading to a file choice.

Description

The DirList Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It calls PickList.Process to perform command processing. It exits on any command that is a PickList exit command, with the possible exception of ccSelect. When ccSelect is executed and the current item is a subdirectory, DirList.Process does not exit; instead, it reads the files in the specified subdirectory and continues processing.

Process calls several user-defined routines. In addition to the routines applicable to the PickList, a DirList calls routines for filename filtering, directory sorting, and filename formatting. See the introduction to this section for more information about these routines.

Example

For a complete example, see the introduction to this section.

See Also

ChangeDirectory
CommandWindow.Process
GetFileName
PickList.Process

Declaration

```
function Reject(var X : DirRec) : Boolean; virtual;
```

Purpose

Return True to exclude a particular file from list.

Description

OPDIR calls this virtual method to determine whether each raw file record (X) should be added to the list of files to display. If Reject returns True, the filename will be ignored and the directory listing will make no mention of the file. DirList's implementation of Reject simply calls another user-defined routine via a procedure pointer.

See the discussion of "Filename Filtering" in the introduction to this section for more information.

See Also

SetRejectFunc

Declaration

```
procedure RemoveMultiFile(N : Word);
```

Purpose

Remove specified file from list.

Description

RemoveMultiFile removes the specified file from all of the DirList's records; at the next call to Draw or Process the file will no longer appear. The call to RemoveMultiFile shifts the items above N down by one in the list. If N specifies the currently highlighted file, the highlight will move to the file following N, unless N is the last file.

This method is useful when writing a full-featured file manager like the one in DESKPOP, where you may have a command to physically delete marked files. If you call RemoveMultiFile to remove each such file then you won't need to force DirList to read, sort, and format the list again. Be careful not to call Process again after removing *all* files, however.

Example

```
repeat
  {...do multi-pick...}
  Dir.InitSequence(N);
  while Dir.HaveSelected(N) do begin
    Dir.RemoveMultiFile(N);
    Dec(N);
    Dir.NextSelected(N);
  end;
until Finished;
```

Removes all selected files from the directory list, then repeats the pick. We decrement the value of N to avoid skipping over selected files. Note that the example doesn't physically erase any files; it just removes them from the directory display.

See RemoveSelectedFiles for an easier way to do this. See GetMultiFile for a complete example of multiple file selections.

See Also

RemoveSelectedFile RemoveSelectedFiles

RemoveSelectedFile / RemoveSelectedFiles

Declaration

```
procedure RemoveSelectedFile;
```

Purpose

Remove current file from pick list.

Description

This method calls RemoveMultiFile with N set to the current file index.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
Dir.SetMask('*.*', AnyFile);
Finished := False;
repeat
  Dir.Process;
  case Dir.GetLastCommand of
    ccSelect : Dir.RemoveSelectedFile;
    ccQuit   : Finished := True;
  end;
until Finished;
```

Removes each selected file entry from the list.

See Also

RemoveMultiFile

Declaration

```
procedure RemoveSelectedFiles;
```

Purpose

Remove all selected files from list.

Description

This method is intended to be used with multi-file pick lists. It removes all marked files from the list by repeatedly calling RemoveMultiFile.

Example

```
Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, MultipleFile);
Dir.SetMask('*.*', AnyFile);
Dir.SetSelectMarker('<', '>');
Finished := False;
repeat
  Dir.Process;
  if Dir.GetSelectedCount = 0 then
    Finished := True
  else
    Dir.RemoveSelectedFiles;
until Finished;
```

Removes all selected files after a multiple file selection.

See Also

RemoveMultiFile

Declaration

```
procedure ResetList;
```

Purpose

Force reread of directory when it may have changed.

Description

This method sets a number of internal DirList flags which indicate that the Draw, PreloadDirList, and Process methods must again read the DOS directory, sort the files, format all filename strings, optionally resize the pick window, reposition the highlight bar, and update the mask header. DirList normally resets the list in this way only when you change the search mask (via SetMask). You may need to reset the list explicitly if your program adds or deletes files, or otherwise updates the directory, without changing the DirList search mask.

DirList provides other methods to remove files from the directory list (see RemoveMultiFile), but it provides none to add files. The RemoveMultiFile approach will be faster than ResetList when it is practical and appropriate.

Example

```
var
  F : File;
begin
  {...Init/setup Dir...}
  Finished := False;
  repeat
    Dir.Process;
    case Dir.GetLastCommand of
      ccSelect :
        begin
          Assign(F, Dir.GetSelectedFile);
          System.Erase(F);
          ResetList;
        end;
      ccQuit :
        Finished := True;
    end;
  until Finished;
end.
```

Selecting a file causes that file to be deleted; ResetList assures that the directory is reread when Process is called again.

See Also

RemoveMultiFile SetMask

SelectedFileInCurDir / SelectItem

Declaration

```
function SelectedFileInCurDir : Boolean;
```

Purpose

Return true if selected file is in current directory.

Description

SelectedFileInCurDir compares the string returned by GetSelectedDir to that returned by Turbo Pascal's GetDir procedure with a drive number of 0. Therefore it returns True if the last file selected in Process is in the current directory of the current drive.

Example

```
{...Pick a file...}
if Dir.SelectedFileInCurDir then
  Fname := Dir.GetSelectedFile
else
  Fname := Dir.GetSelectedPath;
```

If the selected file is in the current directory, it is referred to just by its filename; otherwise, the full pathname is used.

See Also

GetSelectedDir GetSelectedPath

Declaration

```
procedure SelectItem(N : Word); virtual;
```

Purpose

Select an item.

Description

DirList.SelectItem overrides the method of the same name in PickList in order to prevent marking of subdirectories. A call to DirList.SelectItem where N refers to a subdirectory item is simply ignored.

This version of SelectItem makes the DirList behave as described in the "Directory Commands" introduction. However, it means that you can't use a DirList to mark multiple subdirectories for some common operation. If you need to do so, use a PathList object instead, or else derive a new type from DirList that re-overrides the SelectItem method.

Example

```
{...set up for multi-pick...}
repeat
  Dir.Process;
  case Dir.GetLastCommand of
    ccUser0 : for I := 1 to diCurFiles do
                Dir.SelectItem(I);
    ccUser1 : for I := 1 to diCurFiles do
                Dir.DeSelectItem(I);
    ccQuit : Finished := True;
  end;
until Finished;
```

Uses an exit command (ccUser0) to select all displayed files with one keystroke. Subdirectories will not be marked.

See Also

PickList.SelectItem

Declaration

```
procedure SetDirAttr(Color, Mono : Byte);
```

Purpose

Set video attributes for directories.

Description

SetDirAttr simply makes a call to SetPickAttr for the pkAlternate item category in its unselected state. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example

```
Dir.SetDirAttr($02, $70);
```

Displays directory names in green on black for color monitors, reverse video for monochrome monitors.

See Also

PickList.SetPickAttr SetSelectDirAttr

Declaration

```
procedure SetFileAttr(Color, Mono : Byte);
```

Purpose

Set video attributes for files.

Description

SetFileAttr simply makes a call to SetPickAttr for the pkNormal item category in its unselected state. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example

```
Dir.SetFileAttr($06, $70);
```

Displays file names in brown on black for color monitors, reverse video for monochrome monitors.

See Also

PickList.SetPickAttr SetSelectFileAttr

SetMask

Declaration

```
procedure SetMask(Mask : String; FileAttr : Byte);
```

Purpose

Set search mask and attribute and reset list.

Description

SetMask stores a new search mask and search attribute in data fields of the object, then calls ResetList. The next time PreloadDirList, Draw, or Process is called, the DirList will read the DOS directory, sort the list, format the files, and so on.

Mask should be a valid DOS pathname or wildcard combination. (Invalid masks may be specified as well; the DirList will return a non-fatal error after a later call to PreloadDirList, Draw, or Process.) The mask need not specify the current drive or directory. An empty mask is interpreted the same as '*.*'.

FileAttr is a DOS file attribute to match. The DOS unit defines constants that specify the attributes. Normal files have attribute 0; you can add the following constants to specify other attributes.

```
ReadOnly  = $01;
Hidden    = $02;
SysFile   = $04;
VolumeID  = $08;
Directory = $10;
Archive   = $20;
AnyFile   = $3F;
```

Note that DOS uses additive rules in applying these attributes. For example, if you specify FileAttr = ReadOnly, you get read-only files in addition to normal files. If you specify Directory, you get directories and non-directories. You get Archive files whether or not you specify the Archive bit. You can use the DirList's reject function to filter files more precisely.

Examples

```
Dir.SetMask('*.*', AnyFile);
```

Lists all files in the current directory, including subdirectories and special files like System and Hidden files.

```
Dir.SetMask('*.PAS', 0);
```

Lists all files in the current directory that have a .PAS extension; no subdirectories and no special files are included.

See Also

ResetList

Declaration

```
procedure SetRejectFunc(RF : diRejectFunc);
```

Purpose

Set alternate reject function for file selection.

Description

This method specifies the function that Reject will call to determine whether each file should be added to the directory list. Standard reject functions include RejectNone (the default), RejectExtensions, and RejectFiles.

The specified function RF must be compiled under the FAR model, must be global, and must have a parameter list that exactly matches the diRejectFunc type.

See the discussion of "Filename Filtering" near the beginning of this section for more information.

Examples

```
var
  Dir : DirList;
  {$F+}
  function MyRejectFunc(var D : DirRec; DirPtr : DirListPtr) : Boolean;
  begin
    if RejectNone(D, DirPtr) then         {reject "." files and VolumeID}
      MyRejectFunc := True
    else if UpCase(D.Name[1]) = 'A' then  {reject names that start with "A"}
      MyRejectFunc := True
    else
      MyRejectFunc := False;
  end;
  {$F-}
begin
  Dir.Init(25, 5, 55, 20, MaxAvail, PickHorizontal, SingleFile);
  Dir.SetRejectFunc(MyRejectFunc);
  Dir.Process;
  Dir.Done;
end.
```

Installs a custom reject function to exclude all files that begin with the letter 'A'.

```
SetRejectFunc(RejectFiles);
```

Uses a built-in reject function to reject all non-directory files (show only directories).

See Also

AddRejectString Reject

SetSelectDirAttr / SetSelectFileAttr

Declaration

```
procedure SetSelectDirAttr(Color, Mono : Byte);
```

Purpose

Set video attributes for selected directory.

Description

SetSelectDirAttr simply makes a call to SetPickAttr for the pkAlternate item category in its selected state. This specifies the attribute for the pick list highlight bar when it is over a subdirectory name. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example

```
Dir.SetSelectDirAttr($03, $70);
```

Displays selected directories in cyan on black for color monitors, reverse video for monochrome monitors.

See Also

PickList.SetPickAttr SetDirAttr

Declaration

```
procedure SetSelectFileAttr(Color, Mono : Byte);
```

Purpose

Set video attributes for selected file.

Description

SetSelectFileAttr simply makes a call to SetPickAttr for the pkNormal item category in its selected state. This specifies the attribute for the pick list highlight bar when it is over a filename. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example

```
Dir.SetSelectFileAttr($03, $70);
```

Displays selected files names in cyan on black for color monitors, reverse video for monochrome montiors.

See Also

PickList.SetPickAttr SetFileAttr

Declaration

```
procedure SetSortOrder(SP : diSortProc);
```

Purpose

Set alternate routine for sorting.

Description

This method specifies the procedure that SortList will call to sort the files into display order. Standard sort procedures include SortNone (the default), SortName, SortDirName, SortSize, and SortTime.

The specified function SP must be compiled under the FAR model, must be global, and must have a parameter list that exactly matches the diSortProc type.

Calling SetSortOrder also sets internal DirList flags that cause it to resort the list the next time Process is called. Hence, you can change sort order as the result of a Process method exit command and then loop immediately around to call Process again. If you specify SortNone, Process is forced to read the directory from DOS again, since there is no other way to obtain the DOS order.

See the discussion of "Sorting the List" near the beginning of this section for more information.

Examples

```
SetSortOrder(SortNone);
```

Returns to DOS "sort order" by rereading the directory from disk.

```
SetSortOrder(SortDirName);
```

Sort directories first, then sort the files by name/extension.

See Also

SortList

SetXxxFormat / SortList

Declarations

```
procedure SetAllFormat(DirStr, DatePicture, TimePicture : String);
procedure SetNameFormat;
procedure SetNameSizeFormat(DirStr : String);
procedure SetNameSizeKFormat(DirStr : String);
procedure SetNameSizeKTimeFormat(DirStr, DatePicture, TimePicture : String);
procedure SetNameSizeTimeFormat(DirStr, DatePicture, TimePicture : String);
procedure SetNameTimeFormat(DatePicture, TimePicture : String);
procedure SetUserFormat(MaxFmatLen : Byte; FF : diFormatProc);
```

Purpose

Set up for a specified directory format.

Description

DirList supports seven standard directory formats as well as one format that is completely user-defined. The seven standard formats offer added flexibility via specification of date and time picture masks. The formats that support date and time are available only if the UseDates conditional define has been activated in the OPDEFINE.INC file.

Assuming the parameters DatePicture='mm/dd/yy' and TimePicture='hh:mm', the seven standard formats would produce the following output lines:

```
Format                  Width  Example
-------------------------------------------------------------------
SetNameFormat           12     'nnnnnnnn.eee'
SetNameSizeKFormat      19     'nnnnnnnn.eee sssssK'
SetNameSizeFormat       22     'nnnnnnnn.eee sssssssss'
SetNameTimeFormat       27     'nnnnnnnn.eee 01/02/89 13:02'
SetNameSizeKTimeFormat  34     'nnnnnnnn.eee sssssK 01/02/89 13:02'
SetNameSizeTimeFormat   37     'nnnnnnnn.eee sssssssss 01/02/89 13:02'
SetAllFormat            42     'nnnnnnnn.eee sssssssss 01/02/89 13:02 RHSA'
```

The following rules also apply. Subdirectories in NameFormat and NameTimeFormat are indicated by appending '\' to the name. Subdirectories in the other formats are indicated by placing DirStr in the size column. The 'K' used to indicate kilobytes is in uppercase if the diFilesUpcase option is enabled, else lowercase. Kilobytes are 1024 bytes, and are rounded to the next kilobyte. If the DatePicture or TimePicture is empty, the field does not appear in the output. The characters that appear for attributes in AllFormat are controlled by the global typed constants diReadOnlyChar, diHiddenChar, diSystemChar, and diArchiveChar.

Note: The longer the format, the fewer files that can be managed in a given amount of heap space. See the entry for the FileCapacity method and the introductory section on "Filename Formatting" for more information.

See Also

FileCapacity

Declaration

```
procedure SortList; virtual;
```

Purpose

Sort the list of directory entries.

Description

OPDIR calls this virtual method to sort the list of files before formatting and displaying them. DirList's implementation of SortList simply calls another user-defined routine via a procedure pointer. See "Sorting the List" in the introduction to this section for details.

See Also

SetSortOrder

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a directory list in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the DirList to the stream S. It starts by calling PickList.Store then continues to write the data specific to a DirList.

To check for errors that may occur during the Store, call the stream's GetStatus function.

The stream registration procedure for a DirList is DirListStream. There are also a number of procedure pointers that need to be registered independently. The example program below demonstrates how. These registration calls take a little extra effort on your part; in return, Turbo's smart linker is able to strip out the code sections of OPDIR–alternate formats, sorting routines, and so on–that your program doesn't actually use. If we registered all of them for you, the linker would hook them all in.

Because DirList builds onto PickList, most of the PickList's registration issues also apply to DirList. See PickList.Store for details. You don't need to register the user string procedure itself since DirList supplies a specific one and registers it for you.

DirList uses three procedure pointers of its own, for the reject, sort, and format routines; if you are using anything but the default routines (RejectNone, SortNone, NameFormat) you'll need to register the routines you have used. After the example below we list all of the standard procedures and predefined registration codes.

Example

```
uses Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var  Dir : DirList;
     S : BufIdStream;
begin
  {Initialize the directory window}
  Dir.InitCustom(25, 5, 70, 20, DefaultColorSet,
                 DefWindowOptions or wBordered,
                 MaxAvail, PickHorizontal, SingleFile);
  Dir.diOptionsOn(diOptimizeSize);
  Dir.SetPadSize(1, 1);
  Dir.SetAllFormat('<dir>', 'Mm/dd/yy', 'Hh:mmt');
  Dir.SetSortOrder(SortName);

  {Give it a try}
  Dir.SetMask('*.*', AnyFile);
  Dir.Process;

  {Store it to a Stream}
  S.Init('DIRLIST.STM', SCreate, 1024);
  S.RegisterHier(DirListStream);
  S.RegisterPointer(ptPickHorizontal, @PickHorizontal);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.RegisterPointer(ptDirAllFormat, @AllFormat);
  S.RegisterPointer(ptDirSortName, @SortName);
  S.Put(Dir);
  S.Done;
  Dir.Done;
end.
```

Store

Instantiates Dir and stores it to a stream. The example registers the addresses of the list orientation routine (PickHorizontal), the command handler (SingleChoice), the directory format (AllFormat), and the directory sort (SortName). Note the use of the '@' operator to return a pointer to each named procedure or function. Each of the standard support routines provided by OPDIR has a predefined registration code that should be used to avoid conflicts:

Procedure or Function	Pointer Registration Code	Category
RejectNone	ptDirRejectNone	Reject filters
RejectExtensions	ptDirRejectExtensions	"
RejectFiles	ptDirRejectFiles	"
SortNone	ptDirSortNone	Directory sorting
SortName	ptDirSortName	"
SortDirName	ptDirSortDirName	"
SortSize	ptDirSortSize	"
SortTime	ptDirSortTime	"
NameFormat	ptDirNameFormat	File formatting
NameSizeKFormat	ptDirNameSizeKFormat	"
NameSizeFormat	ptDirNameSizeFormat	"
NameTimeFormat	ptDirNameTimeFormat	"
NameSizeKTimeFormat	ptDirNameSizeKTimeFormat	"
NameSizeTimeFormat	ptDirNameSizeTimeFormat	"
AllFormat	ptDirAllFormat	"
DirStringProc	ptDirStringProc	Pick string procedures
MultiStringProc	ptDirMultiProc	"

Note that you'll never need to explicitly register either of the last two routines since OPDIR registers the appropriate one for you.

See the Load example for a program to load this stream.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Load PickList.Store

Declaration

```
procedure UpdateContents; virtual;
```

Purpose

Redraw the contents of the directory window.

Description

You shouldn't need to call UpdateContents directly. Call Draw instead.

UpdateContents performs the following operations:

If no files are available to display, UpdateContents clears the window and exits.

It updates the internal file list (reads DOS directory, sorts, formats, etc.) using internal flags that indicate how much work, if any, it has to do.

It calls PickList.UpdateContents.

See Also

PickList.UpdateContents

2. PathList: Path Selection

A PathList is much like a DirList except that it deals strictly with subdirectories and not with normal files. The PathList lists all subdirectories on a specified drive and allows the user to select one of them. The niftiest feature of PathList is its ability to display the subdirectories in "tree format," which connects all the directories with line-drawing characters in order to make their relationships clear. PathList can also display the list using the plain DOS pathnames if desired.

Except for implementing the line-drawing feature, PathList is much simpler than DirList. It has no need for user-defined reject, sort, and formatting hooks, or for date and time formatting options. Nevertheless, PathList retains enough similarity to DirList that you'll probably want to read about DirList to gain additional background.

Here's an example program that demonstrates the PathList:

```
program PathListExample;  {EXPATH.PAS}
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Path : PathList;
  SelPath : PathStr;
  Result : Word;
begin
  {Make a PathList object with custom window options}
  if not Path.InitCustom(25, 5, 49, 20,            {Window coordinates}
                         DefaultColorSet,          {ColorSet}
                         DefWindowOptions or wBordered, {Window options}
                         8000,                     {Max heap space for files}
                         PickVertical,             {Pick orientation}
                         SinglePath)               {Command handler}
  then begin
    WriteLn('Failed to Init PathList,  Status = ', InitStatus);
    Halt;
  end;
  Path.paOptionsOn(paShowSize+paAltCurDir);
  Path.SetPadSize(1, 1);

  {Pick a directory}
  SelPath := Path.GetPathName('C');
  if SelPath <> '' then
    WriteLn('Selected: ', SelPath)
  else begin
    Result := Path.GetLastError;
    if Result <> 0 then
      WriteLn('Error selecting directory, Result = ', Result)
    else
      WriteLn('No directory selected');
  end;
  Path.Done;
end.
```

Like DirList, the PathList constructor takes a parameter that indicates how much heap space to allocate for an internal array of subdirectory records. Each such record uses 19 bytes, so in this case the Path variable can manage up to 421 subdirectories. (Unlike the DirList, there is no need for a second array of formatted strings.)

Whenever the PathList is used in its default mode of displaying the tree diagram, the list orientation must be PickVertical. Any other orientation may yield very odd looking tree diagrams.

A PathList can be used in either SinglePath or MultiplePath mode. With MultiplePath, you must also call PickList.SetSelectMarker so that markings will be visible.

In this example, we set the PathList options to display the total number of bytes used in each directory (paShowSize) and to color the current directory in the pkAlternate video attribute (paAltCurDir).

The GetPathName method is PathList's equivalent of DirList.GetFileName. Given a specified drive letter, it initializes the internal directory array, displays the pick window, allows the user to make a selection, and returns the selected pathname. GetPathName returns an empty string if any error occurred.

Constants

```
paUpcase          = $0001; {Display directories in uppercase}
paSetCurDir       = $0002; {Position select bar over current directory}
paShowSize        = $0004; {Show the total bytes stored in each directory}
paOptimizeSize    = $0008; {Optimize window size when directory is read}
paAltCurDir       = $0010; {Display current directory in alternate attribute}
```

These are the options that affect the behavior of a PathList. paUpcase forces the directory names to uppercase. If this option is not enabled, directory names are forced to lowercase.

paSetCurDir causes the Process method to set the initial choice to the current directory of the specified drive.

When paShowSize is enabled, PathList displays the total number of bytes in each directory next to the directory name.

paOptimizeSize enables automatic window sizing, so that the pick window shrinks and grows automatically depending on the number of matching directories. The resizing is performed by PickList's OptimizeSize method; see that entry for more details. Note that this option works only if the UseAdjustableWindows define is active.

paAltCurDir causes PathList's string procedure to categorize the drive's current directory as pkAlternate, so that the PickList will display it in a different color.

```
DefPathOptions : Word = paSetCurDir;
```

The default options for a PathList.

```
BadPathOptions    : Word = paReadDir+paFullPathDisp+paUpdating;
```

Options used internally by PathList. The paOptionsOn and paOptionsOff methods will not change these bits of a PathList option word.

```
paTee             = 1;  {┬}
paPipe            = 2;  {│}
paArrow           = 3;  {├}
paBar             = 4;  {─}
paEll             = 5;  {└}
```

Indexes to a PathDrawArray. PathList uses five unique characters to draw the tree diagram and refers to them by these names.

```
paSingleDraw      : PathDrawArray = '┬│├─└';
paDoubleDraw      : PathDrawArray = '╥║╟═╚';
```

Standard arrays of line drawing characters for the tree diagram. PathList uses paSingleDraw by default. You can choose a different character array with the SetDrawingChars method.

Types

```
PathDrawArray     = array[paTee..paEll] of Char;
```

Collects the drawing characters used for the directory tree diagram.

```
PathRec           =              {Data from DOS SearchRec}
  record
    PCat          : Byte;        {Pick category}
    PLevel        : Byte;        {Nesting level of directory. Root = 0}
    PSize         : LongInt;     {Total bytes stored in directory}
    PName         : FileStr;     {Name and extension of file}
  end;
PathRecArray      = array[1..($FFOO div SizeOf(PathRec))] of PathRec;
PathArrayPtr      = ^PathRecArray;
```

The PathList manages an array of PathRec, one for each subdirectory on a given drive. The array stores enough information that PathList can construct the directory tree, the complete pathname of each directory, and the total size of the files there. PathList's constructors allocate as large a PathRecArray as the specified amount of heap space allows. You'll note that a PathRec consumes 19 bytes, so it's easy to figure how many directories you can manage on a given drive.

```
paInitCommandProc = procedure (P : PathListPtr;
                               var StringProc : pkStringProc;
                               var CommandInitProc : pkGenlProc);
```

A procedure type used by PathList constructors to specify a single or multiple choice directory list. OPDIR interfaces the SinglePath and MultiplePath procedures that match this type.

```
PathListPtr = ^PathList;
PathList =
  object(PickList)
    ...
  end;
```

The PathList object.

Declaration
```
destructor Done; virtual;
```

Purpose
Dispose of path list.

Description
This destructor deallocates the heap space PathList uses to store the directory array and then calls PickList.Done.

Example
```
Path.Done;
```
Disposes of the heap space used by Path.

See Also
PickList.Done

Declaration
```
function GetMultiPath(N : Word) : PathStr;
```

Purpose
Return pathname of specified path.

Description
If N is zero or greater than the current number of directories, the empty string is returned. Otherwise GetMultiPath constructs the pathname by tracing the internal array of PathRec. The string includes the drive letter. It is not terminated with a backslash (unless the directory is the root directory).

Example
```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Path : PathList;
  N : Word;
begin
  Path.Init(25, 5, 55, 20, MaxAvail, PickVertical, MultiplePath);
  Path.SetSelectMarker('<', '>');
  Path.Process;
  Path.Erase;
  Path.InitSequence(N);
  while Path.HaveSelected(N) do begin
    WriteLn('Selected: ', Path.GetMultiPath(N));
    Path.NextSelected(N);
  end;
  Path.Done;
end.
```
This complete program shows how to set up and use a multiple choice PathList. It writes out the complete pathname of all selected directories by calling GetMultiPath. Note that many of the methods called are inherited from PickList.

See Also
GetSelectedPath

GetPathName

Declaration

```
function GetPathName(Drive : Char) : PathStr;
```

Purpose

Set drive, pick a path, and return it. Returns empty string if error.

Description

GetPathName is a shell around PathList's Process method. Calling GetPathName makes it easier to select a single pathname for a program. The object that GetPathName references must have already been initialized by calling its constructor. Any customization (formatting and window options) must have already been set up. GetPathName works only with path lists initialized to pick a single directory at a time.

Drive is the letter of the disk drive to choose from ('A', 'B', etc.). If this drive is different than the last specified drive, GetPathName must access the disk in order to read the entire directory structure. For large drives with a significant number of files and directories, this scanning process may take several seconds or more before the user gets any feedback. If this delay is problematic, you'll be better off using the Process method directly, which will give you an opportunity to display a status message while the directory structure is scanned.

If GetPathName returns an empty string, be sure to call GetLastError to classify the error code. The most likely reason for an error is an invalid drive letter.

Example

```
Path.InitCustom(25, 5, 55, 20, 8000, PickVertical, SinglePath);
SelPath := Path.GetPathName('C');
if SelPath <> '' then
  WriteLn('Selected: ', SelPath)
else begin
  Result := Path.GetLastError;
  if Result <> 0 then
    WriteLn('Error selecting directory, Result = ', Result)
  else
    WriteLn('No directory selected');
end;
```

Calls GetPathName to display the directory structure of drive C and allow the user to select a directory.

See Also

PreloadPathList SetDrive

Declaration

```
function GetSelectedPath : PathStr;
```

Purpose

Return selected pathname.

Description

GetSelectedPath works like GetMultiPath, but returns the pathname of only the most recently selected directory.

Example

```
var
  S : PathStr;
...
Path.Process;
Result := Path.GetLastError;
if Result = 0 then
  S := Path.GetSelectedPath
else
  {report error};
```

The string variable S will contain the complete path name of the selected directory.

See Also

GetMultiPath GetPathName

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte;
                 HeapToUse : Word;
                 PickOrientation : pkGenlProc;
                 CommandInit : paInitCommandProc);
```

Purpose

Initialize a path list.

Description

This constructor instantiates a PathList using the default window options (DefWindowOptions), the default color set (DefaultColorSet), the default pick options (DefPickOptions), and the default directory options (DefPathOptions). See InitCustom for more details.

Example

```
if not Path.Init(25, 5, 55, 20,              {Window coordinates}
                 8000,                       {Max heap space for directories}
                 PickVertical,               {Pick orientation}
                 SinglePath)                 {Command handler}
then begin
  WriteLn('Failed to Init PathList,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates Path with default window options and colors.

See Also

InitCustom

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet;
                       Options : LongInt;
                       HeapToUse : Word;
                       PickOrientation : pkGenlProc;
                       CommandInit : paInitCommandProc);
```

Purpose

Initialize a path list with custom window options.

Description

X1, Y1, X2, and Y2 specify the active window coordinates within which the directory list will appear. Colors and Options are interpreted exactly the same as for a PickList. For the purposes of specifying colors, note that the standard PathList display treats all directories as pkNormal items, with the possible exception of the current directory, which it would treat as a pkAlternate item.

HeapToUse is the number of bytes of heap space to allocate for filename storage. InitCustom will allocate at most $FFF1 bytes. If it cannot allocate the lesser of $FFF1 or HeapToUse bytes, the constructor will fail. The maximum number of directories per drive is equal to the heap space allocated divided by 19.

PickOrientation specifies the orientation of the underlying PickList. It can take the value PickVertical, PickHorizontal, or PickSnaking. For PathList displays using the line-drawing tree format, the orientation should always be PickVertical.

CommandInit specifies whether the path list manages multiple directory selections or just a single one. Acceptable values for this parameter are MultiplePath or SinglePath. These values are actually procedures interfaced by OPDIR.

The PickList InitCustom constructor performs most of the work. The pick list's item width is calculated to assure that exactly one column of items will appear.

Beyond that, PathList.InitCustom sets the drive to display to the current default drive. InitCustom does not scan it for directories; it merely sets a flag that indicates the scanning is yet to be done. InitCustom also sets the sector size (for computing directory byte totals) to 1.

The constructor allocates heap space for directory storage and–if the MultiplePath command handler is specified–for the selection BitSet. Note that heap space for the BitSet is in addition to HeapToUse. (It is typically a small amount, however, since the BitSet packs 8 files into each byte: 400 files would fit in a 50-byte BitSet.)

The PickCommands command processor is assigned to the PathList. The object is assigned a unit code of ucDir (for the purposes of identifying it to error and help handlers). See the discussion of "Directory Commands" in the introduction to DirList for more background.

Note that the PathList is assigned the default directory options from DefPathOptions. You can change these by calling paOptionsOn or paOptionsOff after the constructor returns.

If the constructor fails, check the InitStatus variable for a detailed error code.

Example

```
if not Path.InitCustom(25, 5, 55, 20,              {Window coordinates}
                       DefaultColorSet,            {ColorSet}
                       DefWindowOptions or wBordered, {Window options}
                       8000,                       {Max heap space for files}
                       PickVertical,               {Pick orientation}
                       SinglePath)                 {Command handler}
then begin
  WriteLn('Failed to init PathList. Status = ', InitStatus); Halt;
end;
```

Instantiates Path with default colors and a bordered window.

See Also

Init PickList.InitCustom

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a path list from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to PathList.Store. Load first calls PickList.Load to read the underlying PickList. Then it reads data specific to the PathList, including line drawing characters, the selected drive, and option flags.

The stream registration procedure for a PathList is PathListStream. The only procedure pointers that you may need to register independently are those associated with the underlying PickList. See PickList.Load for more details.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var
  Path : PathList;
  S : BufIdStream;
begin
  S.Init('PATHLIST.STM', SOpen, 1024);
  S.RegisterHier(PathListStream);
  S.RegisterPointer(ptPickVertical, @PickVertical);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.Get(Path);
  S.Done;
  Path.Process;
  Path.Done;
end.
```

Instantiates Path from a stream created by the Store example by calling Get, which indirectly calls PathList.Load.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Store PickList.Load

paOptionsAreOn / paOptionsOff / paOptionsOn

Declaration
```
function paOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This function tests the path list options, which are described in the Constant declarations near the beginning of this section.

Example
```
if Path.paOptionsAreOn(paShowSize) then
  {take some action} ;
```
Takes some action if the paShowSize option is on.

Declaration
```
procedure paOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadPathOptions (see the Constants section, above).

Example
```
Path.paOptionsOff(paShowSize);
```
Turns off the option for displaying directory size.

Declaration
```
procedure paOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadPathOptions (see the Constants section, above).

Example
```
Path.paOptionsOn(paShowSize+paUpcase);
```
Turns on options for displaying directories in uppercase and showing the directory size.

Declaration
```
procedure PreloadPathList;
```

Purpose
Read paths in advance.

Description
This method may be called before the window is drawn or a selection is requested via Process. PreloadPathList scans the entire directory structure of the drive specified with SetDrive. A system of internal flags avoids repetition of any part of this work until some external change (such as specifying a different drive) requires it.

This method is especially useful when the drive being scanned is large enough to account for a significant delay. You can write a "please be patient" message prior to calling PreloadPathList and then erase the message before calling Process.

If the selected drive is invalid, PreloadPathList will generate an error, which you can check by calling GetLastError.

Example
```
Path.Init(25, 5, 55, 20, 8000, PickVertical, SinglePath);
Path.SetDrive('C');
FastWrite('Please wait...', ScreenHeight, 1, TextAttr);
Path.PreLoadPathList;
FastWrite('              ', ScreenHeight, 1, TextAttr);
if Path.GetLastError = 0 then
  Path.Process
else
  FastWrite('Invalid drive', ScreenHeight, 1, TextAttr);
```

Displays a status message while scanning drive C.

See Also
SetDrive

Declaration
```
procedure Process; virtual;
```

Purpose
Display list and get one selection.

Description
The DirList Process method follows the general model described under CommandWindow.Process. It calls PickList.Process to get selections once the directory array has been built. If the directory array has not been built previously, there may be a substantial delay while OPDIR scans the specified hard disk.

Example
For a complete example, see the introduction to this section.

See Also
CommandWindow.Process
PickList.Process
GetPathName

ResetList / SetClusterSize

Declaration

```
procedure ResetList;
```

Purpose

Force reread of directories when they may have changed.

Description

This method sets internal PathList flags which indicate that the Draw, PreloadPathList, and Process methods must again scan the specified disk to read all the DOS directories. PathList normally resets the list in this way only when you change the selected drive (via SetDrive). You may need to reset the list explicitly if your program adds or removes subdirectories, or otherwise updates the directory structure, without changing the selected drive.

Example

```
repeat
  Path.Process;
  {...Process exit commands...}
  Path.ResetList;
until Finished;
```

The path list is reset after processing exit commands (in case those exit commands altered the physical directory structure).

See Also

SetDrive

Declaration

```
procedure SetClusterSize(Size : Word);
```

Purpose

Set cluster size for adjusting files to DOS boundaries.

Description

When the paShowSize option is enabled, PathList displays the total bytes used in each subdirectory. By default, it keeps a sum of the exact number of bytes reported for each file by FindFirst/FindNext. In reality, DOS allocates disk space in clusters, so the total space used will be larger than the sum of the reported sizes. SetClusterSize allows you to take this effect into account.

The most typical value to specify for Size is 2048. You can find the actual cluster size for a DOS disk by using DOS.GetDiskInfo.

Note that PathList computes the sizes while it is scanning the directory structure. Therefore you must call SetClusterSize before calling PreloadDirList, Draw, or Process for the first time. Generally you'll want to call SetClusterSize immediately after calling the object's constructor.

Example

```
Path.paOptionsOn(paShowSize);
Path.SetClusterSize(2048);
```

The directory sizes shown in this Path display will not be the sum of the actual file sizes but will instead be the sum (in bytes) of the total number of clusters used in the directory.

See Also

paOptionsOn

Declaration

```
procedure SetDrawingChars(P : PathDrawArray);
```

Purpose

Set the characters used for line graphics.

Description

When the PathList displays the directory in "tree format," it uses an array of line drawing characters to make the tree look nice. By default it uses the single-wide IBM line-drawing set of characters. You can change that by calling SetDrawingCharacters. OPDIR interfaces two typed constants of the appropriate type–paSingleDraw and paDoubleDraw–or you can define others if needed.

Example

```
Path.SetDrawingChars(paDoubleDraw);
```

This path will be drawn using the paDoubleDraw drawing set (all double lines).

See Also

SetFullPathMode

Declaration

```
procedure SetDrive(Drive : Char);
```

Purpose

Specify drive for path list.

Description

When you first initialize a PathList object, it automatically selects the current default drive as the one to analyze and select from. Call SetDrive to specify an alternate drive. If the drive letter is different than the last one specified, SetDrive stores the new drive character and calls ResetList so that the directory array will be updated when next required.

Note that SetDrive does not validate the specified drive. Errors will not be detected until PreloadPathList, Draw, or Process is called.

Examples

```
Path.Init(25, 5, 55, 20, 8000, PickVertical, SinglePath);
Path.paOptionsOn(paShowSize);
Path.SetDrive('C');
Path.Process;
```

This example uses SetDrive to set the current drive to 'C:'. Process then shows the entire directory structure for drive C.

```
SetDrive('C:');
```

Incorrect! The parameter to SetDrive is a single character; no colon is necessary or allowed.

See Also

ValidDrive (OPDOS)

SetFullPathMode / SetLineDrawMode

Declaration

```
procedure SetFullPathMode;
```

Purpose

Display hierarchy with full path names.

Description

PathList supports an alternate display format that may be useful in some situations. Instead of using line-drawing characters and just the unique portions of each directory name, it can display the full pathname of each directory. SetFullPathMode selects an alternate PickList string procedure. The display will not be updated until Draw or Process is called. Note that the window sizing requirements of the full path mode are different than the line drawing mode. (The full path usually requires a wider window.) Typically you should call SetFullPathMode immediately after calling the PathList's constructor.

Example

```
Path.SetFullPathMode;
```

Path will be drawn using full path names instead of the line drawing set.

Declaration

```
procedure SetLineDrawMode;
```

Purpose

Display hierarchy with line draw characters (default).

Description

You don't need to call this routine unless you want a PathList window that supports both the line-drawing and full-path modes. Line-draw mode is the default when you instantiate a PathList window.

Example

```
Path.SetLineDrawMode;
```

Path will be drawn using line-drawing characters.

See Also

SetDrawingChars

Declaration
```
procedure SetPathAttr(Color, Mono : Byte);
```

Purpose
Set video attributes for paths.

Description
SetPathAttr simply makes a call to SetPickAttr for the pkNormal item category in its unselected state. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example
```
Path.SetPathAttr($03, $07);
```
Unselected directories will be displayed in cyan on black on color monitors, white on black for monochrome monitors.

Declaration
```
procedure SetSelectPathAttr(Color, Mono : Byte);
```

Purpose
Set video attributes for selected path.

Description
SetSelectPathAttr simply makes a call to SetPickAttr for the pkNormal item category in its selected state. This specifies the attribute for the pick list highlight bar when it is over a directory name. Color and Mono are the video attributes to be used in color and monochrome video modes respectively.

Example
```
Path.SetSelectPathAttr($0E, $70);
```
Selected directories will be displayed in yellow on black on color monitors, reverse video on monochrome monitors.

Store / UpdateContents

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a list in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the PathList to the stream S. It starts by calling PickList.Store then continues to write the data specific to a PathList.

To check for errors that may occur during the Store, call the stream's GetStatus function.

The stream registration procedure for a PathList is PathListStream. The only procedure pointers that you may need to register independently are those associated with the underlying PickList. See PickList.Store for more details.

Example

```
uses Dos, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpDir;
var  Path : PathList; S : BufIdStream;
begin
  Path.Init(25, 5, 55, 20, 8000, PickVertical, SinglePath);
  Path.paOptionsOn(paShowSize);
  Path.SetDrive('C');
  {Store it to a Stream}
  S.Init('PATHLIST.STM', SCreate, 1024);
  S.RegisterHier(PathListStream);
  S.RegisterPointer(ptPickVertical, @PickVertical);
  S.RegisterPointer(ptPickSingleChoice, @SingleChoice);
  S.Put(Path);
  S.Done;
  Path.Done;
end.
```

Instantiates Path and stores it to a stream. See Load for a program to load this stream.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

Load PickList.Store

Declaration

```
procedure UpdateContents; virtual;
```

Purpose

Redraw the complete pick window.

Description

You shouldn't need to call UpdateContents directly. Call Draw instead.

UpdateContents performs the following operations:

- If no directories are available to display, UpdateContents clears the window and exits.
- It updates the internal directory list using internal flags that indicate how much work, if any, it has to do.
- It calls PickList.UpdateContents.

See Also

PickList.UpdateContents

F. OPMENU - Menu Systems

Fast, intuitive menus and windows are an expected feature of personal computer applications software today. As many of us know, polishing a program's user interface can take more time than writing the core of the application. OPMENU is a powerful collection of objects to reduce the time and drudgery of developing a sophisticated menu system.

Menus come in various styles. Borland's pulldown menus define one commonly used style. Lotus' single-line, horizontally scrolling menu is another. OPMENU easily emulates either of these styles, and lets you define others that fit your fancy. OPMENU also provides integral support for the latest techniques: multi-level nested menus, shadowed frames, exploding submenus, menu hot keys, and mouse support.

OPMENU data structures are fully dynamic--the complexity of a given menu system is limited only by available heap space. OPMENU is implemented by a group of four cooperating objects, shown in the following hierarchy chart along with their ancestor types.

```
[Root]
  [DoubleListNode]
    MenuItemNode
  [DoubleList]
    [CircularList]
      MenuItemList
  [AbstractWindow]
    [RawWindow]
      SubMenu
      [StackWindow]
        [CommandWindow]
          Menu
```

The Menu object is the only one of the four that you'll usually need to declare and use. Even so, we'll document some features of lower level objects that may be useful for unique cases.

Because the Menu is derived from the CommandWindow object, it gains all the functionality of that window type. Inherited features relevant to the Menu include:

- adjustable colors and position
- frame decorations such as headers and shadows
- ability to work in a window stack or on a virtual screen
- CommandProcessor hooks such as keyboard remapping, user exit commands, dynamic exit commands, and background tasks
- context sensitive help hook
- error handling hook

One of the data fields of a Menu object is a MenuItemList, which is a specialized form of linked list whose elements are of type MenuItemNode. Each item that you see in one level of a menu system is a MenuItemNode. The MenuItemNode has data fields that describe the following features of an item:

- displayed item name
- unique identifying number, the "item key"
- optional help string
- screen offset from edge of item list
- offset within item name of an optional highlighted selection character
- protected flag
- pointer to SubMenu

When the pointer to SubMenu is not nil, it means that another list of menu items is attached. The SubMenu object is just a RawWindow with an additional MenuItemList data field and a few overridden methods. Of course, the SubMenu's MenuItemList has MenuItemNodes that themselves may specify attached SubMenus. This multi-way tree structure allows for arbitrarily complex menu systems.

One of the MenuItemNode's data fields deserves further comment, since many of OPMENU's features revolve around it. That field is the "unique identifying number" which, for historical reasons, we refer to as the item key. Each item in a menu system–no matter how complex the system–must be assigned a unique number at the time the system is built. To highlight a default item in a menu, you must specify its key. When the user selects an item (via the Process method), OPMENU returns its unique key. Most menu-based applications will use a Case statement immediately following the call to Menu.Process; each case label will be an item key, and the label's statement will perform actions specific to that item.

The data type of an item key is LongInt, so you need have no fear of running out of codes. Item keys are typically assigned in ascending numerical order, starting with 1, but there is no requirement for the keys to be sequential. Keep the following potential constraint in mind, however. Many applications provide context-sensitive help for each menu item. If you use the OPHELP unit to display this help, you'll probably want OPHELP's help topic number to be the same as the menu item key. Valid topic numbers for OPHELP are limited to the range 1..65535. Also, OPHELP uses space more efficiently when there are no unused topic numbers between 1 and the highest one. Of course, it's easy enough to provide an intermediate mapping array (item key to topic number), so even the use of OPHELP imposes no fundamental limitations on the menu system.

Building a Menu System

How is a particular menu system initialized? In **Object Professional**, there are two ways: you can initialize the system with a series of source statements, which starts with a constructor call for the Menu object and continues with statements that add each submenu and menu item; or you can read the menu from a binary library using just one statement (plus a few to open and close the library file).

The library approach has the advantage of keeping the source code simple. For complex menu systems, the executable code size will often be smaller as well. Disadvantages include the need for a separate library file or a separate compilation step to link the library into the EXE; and source code that doesn't document itself.

Either way you choose, the supplied utility MAKEMENU (described in section H of this chapter) will save you a lot of work. MAKEMENU lets you design a menu system visually and interactively, then writes out the menu in source form, library form, or both. You'll probably use MAKEMENU to design all of your menu systems, since it relieves you of the drudgery of computing window coordinates, item positions, video attribute values, and all the other niggling details. MAKEMENU also makes it easy to modify your menu system as it grows and matures.

Even so, you should have a basic understanding of the methods used to build a menu system. When you do, you'll be able to tweak MAKEMENU's code and to use advanced features that MAKEMENU cannot access. With that in mind, let's look at a simple example program that builds a menu system, displays it, and gets a selection.

```
program MenuExample;  {EXMENU1.PAS}
uses
  Dos, OpInline, OpString, OpRoot, OpCrt, OpFrame, OpCmd, OpWindow, OpMenu;
{Menu item constants}
const
  miFirst         = 1;
  miAlpha         = 3;
  miBeta          = 4;
  miSecond        = 2;
  miGamma         = 5;
  miDelta         = 6;

function InitMenu(var M : Menu) : Word;
  {-Initialize menu system generated by MAKEMENU}
const
  {Frame constants}
  Frame1 = '╒╤╛═||';
  Frame2 = '╓╥╜═||';
begin
  with M do begin
    if not InitCustom(2, 2, 16, 2, DefaultColorSet,
                      wBordered+wClear+wUserContents,
                      Horizontal) then begin
      InitMenu := InitStatus;
      Exit;
    end;
    mnOptionsOn(mnAlphaMatch+mnSelectOnMatch+mnArrowSelect+mnAllHotSpots);
    mnOptionsOff(mnAllowPending+mnPopOnSelect+mnUseItemForTopic+
      mnSelectOnClick);
    wFrame.SetFrameType(Frame1);
    AddItem('First', 2, 1, miFirst);
      AddFramedSubMenu(3, 4, 9, 5, Vertical, Frame2);
      AddItem('Alpha', 1, 1, miAlpha);
      AddItem('Beta', 2, 1, miBeta);
      ItemsDone;
    AddItem('Second', 9, 1, miSecond);
      AddFramedSubMenu(9, 4, 15, 5, Vertical, Frame2);
      AddItem('Gamma', 1, 1, miGamma);
      AddItem('Delta', 2, 1, miDelta);
      ItemsDone;
    ItemsDone;
    InitMenu := GetLastError;
  end;
end;

var
  Status : Word;
  M      : Menu;

begin
  Status := InitMenu(M);
  if Status <> 0 then begin
    WriteLn('Error initializing menu: ', Status);
    Halt(1);
  end;

  M.Draw;
  M.Process;
```

```
    {Skeleton for menu choice handling}
    if M.GetLastCommand = ccSelect then
      case M.MenuChoice of
        miAlpha : {do something};
        miBeta  : {};
        miGamma : {};
        miDelta : {};
      end;

    M.Erase;
    WriteLn('Item ', M.MenuChoice);
    WriteLn('Command ', M.GetLastCommand);
    M.Done;
  end.
```

This program is a slightly simplified version of one generated by MAKEMENU. You'll note that it starts with a list of menu item constants--the item keys referred to in the previous section. We recommend that you always refer to these keys by symbolic name rather than directly by number. Doing so makes it much easier to extend and maintain the menu system. MAKEMENU automatically generates such symbolic names for you, or allows you to assign names of your own.

The InitMenu function builds the menu system by initializing the var parameter M. (MAKEMENU generates the InitMenu function this way because it doesn't know what name you want to give to your actual Menu variable. InitMenu initializes whatever menu variable you pass as its parameter.) The call to the InitCustom constructor specifies the coordinates of the top level menu's window, its colors and options, and the orientation of the top level item list--Horizontal or Vertical. If the constructor fails (out of memory, invalid coordinates, etc.), InitMenu returns the status code.

The calls to mnOptionsOn and mnOptionsOff specify other details of the menu system's behavior. We'll describe these options in the Constants section below. SetFrameType specifies the characters used for the top level menu's border. Calls to other methods inherited from RawWindow or Frame can go here as well--to add headers, shadows, explosions, and so on.

So far the Menu's initialization has been just about like any other window; the call to AddItem enters new territory. AddItem instantiates a MenuItemNode object and links it into the top level menu's MenuItemList. This particular call adds a menu item named 'First', whose display offset (columns from the left edge of the window) is 2, selection character is at offset 1 in the name ('F'), and item key is miFirst (which equals 1).

The item 'First' has a submenu attached to it. We indicate this by the call to AddFramedSubMenu. This call instantiates a SubMenu object and links it into First's submenu pointer field. AddFramedSubMenu's parameters specify the new submenu's window coordinates, the orientation of its item list, and the characters used for its border.

The next calls to AddItem apply to the submenu, not to the top level menu. MAKEMENU indents the source code it generates in order to make this relationship clear. Items continue to attach to the nested submenu until ItemsDone is called, after which AddItem applies to the previously active submenu (in this case the top level menu). The example adds a 'Second' item to the top level menu, hangs a submenu on that item, and continues in the same vein until it has called ItemsDone for the top level menu.

InitMenu ends by returning the value of GetLastError, which will return zero if no errors occurred during the initialization.

The example's main program block starts by calling InitMenu to initialize the Menu instance variable M. After assuring that InitMenu was successful, it calls Menu's special version of the Draw method. Because the menu hasn't been displayed before, Draw simply draws the top level menu and highlights the default item (which is the first item unless you specify otherwise).

Calling Process hands keyboard control to the user, who can browse around the menu system with the arrow keys, <Enter>, and <Esc>. When the user presses <Enter> while the highlight bar is on an item without a submenu, Process exits. As in other CommandWindow derivatives, the last command is retrieved by GetLastCommand; checking this value assures that the user didn't press <Esc> (at the top level of the menu) or some other exit command.

MenuChoice returns the item key of the selected item. We use a case statement to select among the various item keys, then perform some action specific to that item. You'll note that the miFirst and miSecond item keys are not in the case statement. That's because those items have submenus and therefore will never exit Process with a command code of ccSelect. (In special cases, they might exit with a different command, however.)

The example concludes by erasing the menu system and disposing of it. (Menu.Erase overrides the window's usual Erase method and will erase all active submenus, not just the top level menu.)

In the sections that follow, we'll describe many ways to extend these basic capabilities.

Menu Customization

The previous example showed us how to do a nice vanilla pulldown menu system. OPMENU provides plenty of ways to make fancier menus. In this section we'll summarize many of the techniques for customizing the appearance of a menu system. You can refer to the method descriptions later in the chapter for the details of using each technique. Unless otherwise specified, MAKEMENU allows you to adjust these features and generates the appropriate source code.

First, remember that the Menu object is derived from CommandWindow, and thus inherits all the capabilities of that object. If you don't find what you're looking for in the discussion that follows, look back to the CommandWindow, RawWindow, and Frame objects.

In the example above, we showed the default way to add items to a submenu. There are several methods besides AddItem, each of them with advantages in specific situations.

AddItemPtr, unlike AddItem, does not allocate heap space to store the item's name string. Instead it simply stores the address of the string passed to it. This approach saves memory because it avoids creating a duplicate copy of the name. (The literal string you normally pass to AddItem is stored in the code segment; AddItem duplicates that string on the heap.) AddItemPtr requires that you declare the item name strings as typed constants. (The compiler won't let us use the address of a literal string in an executable section of the program.) The typed constants create less readable source code, but you may find AddItemPtr worthwhile if memory is tight and you have a large menu system.

AddItemHelp takes an additional parameter, a help string. OPMENU doesn't do anything with this string but store and return it. Even so, the string can be very useful in combination with the "current item hook." In the section describing that hook we'll show how to display a continuously updated context-sensitive help window using each item's help string.

AddItemHelpPtr combines the capabilities of AddItemPtr and AddItemHelp. Neither the name string nor the help string are stored on the heap, cutting duplicated memory usage.

AddSeparator adds a special kind of item: a bar that spans the menu's window and separates one group of items from the next. Of course, OPMENU will never let the selection highlight land on the separator bar. In fact, if you look behind the scenes, you'll find that AddSeparator doesn't add a menu item at all; instead it adds a span header (see the Frame object). But you call it much like any other AddItem method, so we mention it here.

AddSeparatorColor is just like AddSeparator, but you can specify non-default colors.

Similarly, there are variations on the AddFramedSubMenu call to add a submenu. AddSubMenu attaches an unbordered submenu (appropriate for a Lotus-style menu system). AddSubMenuCustom takes additional parameters that allow you to specify a custom ColorSet and custom window options for the submenu.

A variation on the ItemsDone routine, ItemsDoneCustom, takes two parameters. If the first parameter, OptimizeSize, is True (and the UseAdjustableWindows define is active), ItemsDoneCustom automatically resizes the associated submenu window to just surround the items. The second parameter, DefaultKey, specifies which item should be the initial default when the menu is activated. MAKEMENU won't generate calls to ItemsDoneCustom; if you need to set a default item besides the first in each list, you'll need to change MAKEMENU's code to call ItemsDoneCustom or else use another method, DefaultItem, described later.

The next group of methods we'll summarize is for changing characteristics of the current submenu or item list while the menu system is being built. Calls to these methods would be inserted at the locations shown in the following code fragment.

```
  with M do begin
    if not InitCustom(...) then begin
      InitMenu := InitStatus;
      Exit;
    end;
    wFrame.SetFrameType(Frame1);
--> insert customizations here
    AddItem('First', 2, 1, miFirst);
      AddFramedSubMenu(3, 4, 9, 5, Vertical, Frame2);
----> insert customizations here
      AddItem('Alpha', 1, 1, miAlpha);
```

Customizations inserted at the first marked location would apply to the top level menu only. Those inserted at the second location would apply to the submenu attached to the item 'First'.

Several of the methods—SetNormAttr, SetSelectAttr, SetHighAttr, and SetProtectAttr—are used to specify the display colors of the items. Norm is for unhighlighted characters of unselected items, Select is the color of the selection bar which moves with the cursor keys, High highlights the optional character used to select an item alphabetically, and Protect applies to protected items. SetFrameAttr, SetHeaderAttr, and SetShadowAttr specify the colors for the portions of the submenu frame that differ from the top level menu.

SetLeftPad specifies the number of window background characters appended to the left side of each item. This option applies to vertical lists only.

SetSelectMarker specifies strings that are concatenated to the left and right of the currently selected item, augmenting the selection bar.

Because each submenu is derived from the RawWindow object, it can have headers, shadows, and open/close procedures for special effects like explosions. The Menu object overrides some of the RawWindow and Frame methods in order to make these features more easily accessible.

The Menu versions work like the originals, except that they automatically apply to the main frame (wFrame) of the current submenu and they check the Menu's error variable before they proceed. This error check leads to a major convenience: you don't need to check for errors after each step in building a menu. Menu provides versions of the following methods: AddHeader, AddHeaderColor, AddCustomHeader, AddShadow, AddShadowColor, EnableExplosions, and EnableNormalOpen.

Another group of customization methods is used to modify the appearance of an entire menu system. Instead of working on a single submenu, these methods automatically change every submenu as well as the main menu. You can make calls to these methods at any time after the menu system has been completely initialized (at the end of InitMenu, or after it returns).

The following methods modify the color settings: SetAllNormAttr, SetAllSelectAttr, SetAllHighAttr, SetAllProtectAttr, SetAllFrameAttr, SetAllHeaderAttr, SetAllShadowAttr, and SetHelpAttr. Additional methods modify other window properties: AddShadows, EnableAllExplosions, EnableAllAlternateOpen, EnableAllNormalOpen, and SetAllSelectMarker.

MAKEMENU doesn't generate calls to these "global" routines. It produces the same results by building the menu system with the correct properties in the first place. The most common application of these routines is to user-configurable programs: perhaps the user can enable and disable shadows or explosions in the menu system, for example.

To provide a better feel for some of these options, let's take the introductory example program and add some fancy features to it (using MAKEMENU), then look at the InitMenu routine again.

```
function InitMenu(var M : Menu) : Word;
  {-Initialize menu system generated by MAKEMENU}
const
  {Frame constants}
  Frame1 = '╒╕╛═╡╞';
  Frame2 = '╓╖╜═╡╞';
begin
  with M do begin
    if not InitCustom(2, 2, 16, 2, MenuColors,
                      wBordered+wClear+wUserContents,
                      Horizontal) then begin
      InitMenu := InitStatus;
      Exit;
    end;
    mnOptionsOn(mnAlphaMatch+mnSelectOnMatch+mnArrowSelect+mnAllHotSpots);
    mnOptionsOff(mnAllowPending+mnPopOnSelect+mnUseItemForTopic+
      mnSelectOnClick);
    wFrame.SetFrameType(Frame1);
    AddHeader('Top Level', heTC);                                {***}
    AddItem('First', 2, 1, miFirst);
      AddFramedSubMenu(3, 4, 11, 6, Vertical, Frame2);
      AddSeparator('╞', '═', '╡', 2);                            {***}
      SetSelectMarker('>', '<');                                 {***}
      AddItem('Alpha', 1, 1, miAlpha);
      AddItem('Beta', 3, 1, miBeta);
      ItemsDone;
    AddItem('Second', 9, 1, miSecond);
      AddFramedSubMenu(9, 4, 15, 5, Vertical, Frame2);
      SetNormAttr(LtCyanOnBlue, LtGrayOnBlack);                  {***}
      SetSelectAttr(BlackOnLtGray, BlackOnLtGray);               {***}
```

```
      SetHighAttr(LtRedOnBlue, WhiteOnBlack);                {***}
      EnableExplosions(15);                                  {***}
      AddShadow(shBR, shOverWrite);                          {***}
      AddItem('Gamma', 1, 1, miGamma);
      AddItem('Delta', 2, 1, miDelta);
      ItemsDone;
    ItemsDone;
    InitMenu := GetLastError;
  end;
end;
```

Lines added this time are emphasized with a {***} comment. We've added a header, 'Top Level', to the top level menu. The 'First' submenu has a separator bar between the 'Alpha' and 'Beta' items. The selected item on this submenu will be surrounded with angle brackets, e.g., '>Alpha<'. The 'Second' submenu provides explosion special effects when it opens and closes, and also has a shadow. It uses custom colors for normal and selected items, and for the highlight character.

We'll close this section with a brief mention of three other methods that may be useful to advanced users of **Object Professional**. We use these routines to implement special features in MAKEMENU and to write other methods in the Menu object itself. VisitAllItems performs a depth-first traversal of the entire menu system. For each item it encounters, it calls a user-written procedure and passes a pointer to the MenuItemNode, a pointer to the menu system, and an untyped var parameter containing user-specified data. VisitAllWindows and VisitAllItemLists perform similar traversals for each window object and each MenuItemList in a menu system. See the method descriptions for more details.

Menu Commands

Menu's Process method offers appropriate commands to navigate the items and submenus. The list below arranges them by category. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant MenuKeySet (in OPMENU.ICD).

Cursor Movement

ccLeft **<Left>, <CtrlS>**

Highlight bar moves left one item on a horizontal submenu. If the bar is on the leftmost item of the submenu, it wraps to the rightmost item. If a) the active submenu has a vertical orientation, b) the mnArrowSelect option is enabled, c) the submenu's parent submenu has a horizontal orientation, and d) the parent has additional unprotected items, then the current submenu is erased and the parent's highlight bar is moved left to the next item. If that item has a submenu, the submenu is selected and its default item becomes active. If the parent item has no submenu and the mnAllowPending option is enabled, OPMENU sets an internal flag that reminds it to select the next available submenu if the ccLeft (or ccRight, ccHome, or ccEnd) command is executed again.

ccRight **<Right>, <CtrlD>**

Highlight bar moves right one item on a horizontal submenu. Additional behavior is analogous to that of ccLeft (with right replacing left).

ccHome **<Home>, <CtrlQ><S>**

Highlight moves to first item of horizontal or vertical submenu. If a submenu is pending (see ccLeft), it is selected.

ccEnd **<End>, <CtrlQ><D>**
Highlight moves to last item of horizontal or vertical submenu. If a submenu is pending (see ccLeft), it is selected.

ccUp **<Up>, <CtrlE>, <CtrlW>**
Highlight moves up one row on a vertical submenu. Additional behavior is analogous to that of ccLeft (with up replacing left and vertical replacing horizontal).

ccDown **<Down>, <CtrlX>, <CtrlZ>**
Highlight moves down one row on a vertical submenu. Additional behavior is analogous to that of ccLeft (with down replacing left and vertical replacing horizontal).

Other

ccSelect **<Enter>**
If the current item has no submenu, Process exits and MenuChoice returns the key of the current item. If the current item has a submenu, display the submenu and make its default item current.

ccMouseSel **<ClickLeft>**
Process scans the stack of active submenus (from the most recently selected submenu back to the top level menu). If the mouse position corresponds to a hot spot (see OPFRAME) of the current submenu or to a hot spot of another active submenu and the mnAllHotSpots option is enabled, Process exits, leaving the last command code set to ccMouseSel. If the mouse position corresponds to an unprotected item in the current submenu, the highlight bar is moved to that item. If the item was already highlighted or if the mnSelectOnClick option is enabled, Process changes the command code to ccSelect and exits. If the mouse position corresponds to an unprotected item in another active submenu, submenus are erased until the indicated item in that submenu is current. If the mnSelectOnClick option is on, Process changes the command code to ccSelect and exits. Believe it or not, this behavior is quite intuitive when you try it instead of reading about it! This command is available only if the UseMouse conditional define has been activated.

ccQuit **<Esc>, <ClickRight>**
If the current submenu is the top level menu, Process exits with the ccQuit command code (without erasing the top level menu). Otherwise, the current submenu is erased and its parent's active item becomes current.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling MenuCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. Menu will pass the help procedure the following three parameters: ucMenu, @Self, and a help topic number. If the mnUseItemForTopic option is enabled, the help topic number is the same as the current item key. Otherwise, the help topic is the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

ccChar **any alphanumeric character**
If the mnAlphaMatch option is enabled, the current item list is searched for an item whose SelectChar method result matches the character (case-insensitive). Searching starts on the next item beyond the current one. If a match is found, the selection bar is moved to that item. If the matching item has a submenu, the submenu is displayed. If the mnSelectOnMatch option is enabled and the item has no submenu, Process changes the command code to ccSelect and exits.

Mouse Support

Mouse support is built into the Menu object; all you need to do is activate it. To try it, add the following lines to the introductory example, just after the call to InitMenu successfully completes.

```
if MouseInstalled then begin
  with DefaultColorSet do
    SoftMouseCursor($0000, (ColorMono(MouseColor, MouseMono) shl 8)+$04);
  MenuCommands.cpOptionsOn(cpEnableMouse);
end;
```

This code first assures that a mouse is installed. If so, it defines the mouse cursor characteristics. This step is optional since a software mouse cursor is the default, but you'll probably want to set its colors to be consistent with the rest of your ColorSet. You should always use a software (as opposed to hardware) mouse cursor with the Menu object. See OPMOUSE for more information about defining the mouse cursor.

The call to MenuCommands.cpOptionsOn initializes the Menu's command processor to support the mouse. See OPCMD for more information, especially if you are using background task hooks.

The Menu object provides no support for scroll bars, since menu item lists don't scroll. There's nothing stopping you from adding scroll bars to each window of a menu system, but Menu won't maintain the slider position or interpret clicks on the scroll bar. (If you have need for an item list that contains more items than will fit in the window, use a PickList object that is activated by the selection of a Menu item. Your end-user won't be able to tell where the Menu leaves off and the PickList begins.)

Drawing and Erasing Menu Systems

The Menu object, like all other window derivatives in **Object Professional**, supports the Draw and Erase methods. Because a menu system is usually a collection of related windows rather than just a single one, the behavior of Draw and Erase has been modified appropriately. Several additional routines are also available for finer control of the menu's appearance.

When you call Draw for the first time, the top level menu's frame (border, shadows, headers) is displayed, then its item list is drawn. If the mnMainSelect option is enabled, the current item will be marked with the "select" video attribute and with selection markers if enabled. If the mnMainHighlight option is enabled, the alpha search characters of each item will be highlighted. The availability of these options allows you to display the top level of a menu system in a "dormant" state until the user actually accesses it.

When you call Menu's Process method, the current item is marked and the alpha search characters are highlighted, and then the user can browse around the system and select an item. When Process exits, it does *not* erase the menu system (with one possible exception: if the mnPopOnSelect option is enabled, the submenu of the selected item will be erased automatically). It is your application's responsibility to decide whether to erase the menu system when Process returns. The Menu object offers several ways to erase it.

The Erase method removes the entire menu system from the screen. However, Menu retains an internal stack of which submenus were active. When you call Draw or Process again, all previously active submenus will be redrawn.

EraseClear also erases the entire menu system. However, the internal stack is cleared and the next call to Draw or Process will draw just the top level menu.

EraseAllSubMenus erases all but the top level menu. This method also takes two parameters, ShowSelect and ShowHighlight, which let you specify whether the selection bar and alpha search highlights remain visible.

Finally, EraseCurrentSubMenu erases just the submenu that was most recently active. It will not erase the top level menu.

You can also control exactly how the menu system will be drawn the next time it appears on the screen. The DefaultItem method specifies that a particular item key will become the default item of its submenu. DefaultPath preselects all of the submenus leading to a specified item; the next time the menu system is drawn, all of those submenus will appear.

Whether the menu system is already on the screen or not, you can update the menu's appearance by using other methods to select a different item. SelectItem redraws the menu system to make a specified item (anywhere in the system) current. SelectSubMenu redraws the menu to make the default item of a particular submenu current. (This routine is useful for menu systems that emulate Borland's integrated development environment. If you press the <AltF> key in that environment, the Files submenu will be displayed with the most recently selected item highlighted.) SelectItemByPos selects the item, if any, at a specified screen position; this method is useful when a different (non-menu) window is current and the user mouse-clicks on a menu bar that is visible elsewhere on the screen.

Finally, the Redraw method encapsulates a call to Erase and Draw after temporarily redirecting video output to a virtual screen. This method is useful after one or more menu items have been changed (to protect or unprotect them, or to change user-defined status information).

Access Methods

Menu provides direct access to many of the methods of the objects it depends upon–Frame, RawWindow, MenuItemNode, MenuItemList, SubMenu, etc.–by overriding the appropriate method names and performing the appropriate data field reference, pointer reference, or typecast. This approach makes it easier for you to read and customize the code that MAKEMENU generates.

However, we can't do the same thing for every method without bloating the Menu object beyond reason. Instead, Menu offers several "access methods":

```
function ActiveWinPtr : RawWindowPtr;
  {-Return pointer to active raw window}
function ActiveListPtr : MenuItemListPtr;
  {-Return pointer to active item list}
function ActiveSubPtr : SubMenuPtr;
  {-Return pointer to active submenu}
function ActiveItemPtr : MenuItemNodePtr;
  {-Return pointer to active item}
function FindItem(Key : LongInt) : MenuItemNodePtr;
  {-Return pointer to specified item node}
```

Remember that a menu system is a tree-like data structure. At any point there is an active item, an active window, an active submenu, and an active item list. By using the access methods, you can reach the data fields and methods of the active object. For example, suppose that you wanted to dynamically change a header of the active window. You would use a statement like the following:

```
M.ActiveWinPtr^.ChangeHeader(HdrNum, NewHdrStr);
```

where M is the Menu variable, ChangeHeader is a method of the RawWindow object, and HdrNum and NewHdrStr are appropriate variables or constants in your program.

Another example: suppose you want to retrieve the help string of the active item. The following statement does so:

```
HelpStr := M.ActiveItemPtr^.HelpString;
```

The final access method, FindItem, searches the menu system to find the specified item key and returns a pointer to its MenuItemNode. Using this pointer, you can call any of the MenuItemNode methods. For example,

```
var
  IPtr : MenuItemNodePtr;
...
IPtr := M.FindItem(10);
if IPtr^.SubPtr <> nil then
  {Item 10 has a submenu};
```

If no item is found with the specified key, FindItem will return nil.

Protected Menu Items

A protected menu item is drawn in a separate video attribute, and the user cannot move the selection bar to it. Such items are useful in at least two situations. In the first case, items must be protected and unprotected dynamically. For example, it doesn't make sense to save a file when none is loaded, or to delete a record when none has yet been selected. MAKEMENU and MAKESCRN use this capability extensively to disable particular commands when they aren't applicable. In the second situation, items are protected statically, that is, throughout the entire run of an application. For example, you might use certain items in a menu as informative subtitles, never to be selected.

OPMENU has several methods to support these capabilities. First, the Menu object offers the ProtectItem and UnprotectItem methods. Given the key of a specified item, these methods find the item's MenuItemNode pointer, then set or clear its protection flag. There is no immediate screen update (you might be changing more than one item), but you can call Menu.Redraw to update the screen if the menu is already visible. ProtectItem also checks to see whether the current item of any submenu is being protected; if so, ProtectItem attempts to select a nearby unprotected item.

To see a quick example of a protected item, add the following line to the introductory example program, just prior to the M.Draw call. When you run the program, you'll see that the 'Delta' item can no longer be selected.

```
M.ProtectItem(miDelta);
```

For cases where many items are being protected and unprotected frequently, the overhead of searching the menu tree to find each item may be too high. That was the case in MAKEMENU and MAKESCRN. To avoid the overhead, we stored a pointer to each MenuItemNode, and then called each object's Protect and Unprotect methods directly. There are several ways to get pointers to items. One way is the FindItem method, just discussed in the previous section. Another way is to grab the item's pointer when the menu is built. For example, to store the 'Delta' item's pointer, we'd add the following line to the InitMenu function:

```
      AddItem('Delta', 2, 1, miDelta);
----> DeltaPtr := ActiveItemPtr;
```

This approach requires no searching.

The final way is to use the VisitAllItems method (discussed in "Menu Customization" above) with a user-defined routine that stores each pointer as it is visited. We use this technique in MAKEMENU, where we store the address of every menu item in an array of pointers to achieve the fastest possible access. See the statement

```
{set up pointers to menu items for fast protect capability}
MainMenu.VisitAllItems(SetItemPointers, Dummy);
```

in MMMAIN.IN1 for a complete example.

The Custom String Procedure

Many menus include not only items to choose from, but also status about the items. For example, a menu selection might toggle word-wrap in a text editor; you'd want to show the user whether word-wrap was On or Off, and a good place for that would be immediately adjacent to the menu item.

OPMENU allows for this possibility by calling a user-defined routine while preparing each item for display. By default, it calls a routine that does nothing (NoCustomization, interfaced by OPMENU). You can enable a routine of your own by calling Menu's SetCustomStringProc method:

```
procedure Menu.SetCustomStringProc(CP : mnCustomProc);
  {-Set item customization procedure for entire menu system}
```

The parameter passed to SetCustomStringProc must be a procedure that is compiled under the FAR model (using the {$F+} compiler directive), is global, and matches the following prototype declaration:

```
{$F+}
procedure CustomStringProc(var Name : String; Key : LongInt;
                           Selected, Highlighted : Boolean;
                           WPtr : RawWindowPtr);
```

It's actually possible to have a separate custom string procedure for each MenuItemList. The MenuItemList also has a method called SetCustomStringProc that works just like Menu's but applies only to the specified item list.

Prior to calling the custom string procedure, OPMENU initializes the Name parameter with the item's fixed name string, concatenates the selection markers (if any), and prepends left padding characters (if any). If the associated item list is vertically oriented, OPMENU pads Name to the full width of the window. OPMENU passes the item's Key to the string procedure to identify it for appropriate customization. If the item is going to be marked with the selection bar, Selected is True. If alpha search highlighting is active, Highlighted is True. OPMENU also passes the address of the associated window; the routine could use this pointer to determine the window width, position, or any other needed information.

The custom string procedure's job is to modify the Name parameter to provide status information. For a vertical item list, it should not modify Name's length; instead it should overwrite the existing padding characters within the string. String customization is not usually performed on horizontal item lists, but if you want to do so, you should append the information to the initial Name string.

Let's add a custom string procedure to the introductory example. We'll just show 'On' or 'Off' for each menu item, and toggle that state whenever the item is selected.

```
program MenuExample;  {EXMENU2.PAS}
uses
  Dos, OpInline, OpString, OpRoot, OpCrt, OpFrame, OpCmd, OpWindow, OpMenu;
{Menu item constants}
const
  miFirst         = 1;
  miAlpha         = 3;
  miBeta          = 4;
  miSecond        = 2;
  miGamma         = 5;
  miDelta         = 6;

var
  AlphaOn : Boolean;
  BetaOn : Boolean;
  GammaOn : Boolean;
  DeltaOn : Boolean;

{$F+}
procedure CustomStringProc(var Name : String; Key : LongInt;
                           Selected, Highlighted : Boolean;
                           WPtr : RawWindowPtr);
const OffOn : array[Boolean] of String[3] = ('Off', ' On');
var   On : Boolean;
begin
  case Key of
    miAlpha : On := AlphaOn;
    miBeta  : On := BetaOn;
    miGamma : On := GammaOn;
    miDelta : On := DeltaOn;
    {don't customize the rest of the items}
    else      Exit;
  end;
  Move(OffOn[On][1], Name[Length(Name)-2], 3);
end;
{$F+}

function InitMenu(var M : Menu) : Word;
  {-Initialize menu system generated by MAKEMENU}
const
  {Frame constants}
  Frame1 = '╒╘╛═╡╡';
  Frame2 = '╤╙╛═╡╡';
begin
  with M do begin
    if not InitCustom(2, 2, 16, 2, DefaultColorSet,
                      wBordered+wClear+wUserContents,
                      Horizontal) then begin
      InitMenu := InitStatus;
      Exit;
    end;
    mnOptionsOn(mnAlphaMatch+mnSelectOnMatch+mnArrowSelect+mnAllHotSpots);
    mnOptionsOff(mnAllowPending+mnPopOnSelect+mnUseItemForTopic+
      mnSelectOnClick);
    wFrame.SetFrameType(Frame1);
    AddItem('First', 2, 1, miFirst);
      AddFramedSubMenu(3, 4, 13, 5, Vertical, Frame2);
      AddItem('Alpha', 1, 1, miAlpha);
      AddItem('Beta', 2, 1, miBeta);
      ItemsDone;
```

```
    AddItem('Second', 9, 1, miSecond);
      AddFramedSubMenu(5, 4, 15, 5, Vertical, Frame2);
      AddItem('Gamma', 1, 1, miGamma);
      AddItem('Delta', 2, 1, miDelta);
      ItemsDone;
    ItemsDone;
    InitMenu := GetLastError;
  end;
end;

var
  Status : Word;
  M      : Menu;

begin
  Status := InitMenu(M);
  if Status <> 0 then begin
    WriteLn('Error initializing menu: ', Status);
    Halt(1);
  end;

  {Initialize state variables}
  AlphaOn := False;
  BetaOn := False;
  GammaOn := False;
  DeltaOn := False;

  {Enable the custom string procedure}
  M.SetCustomStringProc(CustomStringProc);

  repeat
    M.Process;
    if M.GetLastCommand = ccSelect then
      case M.MenuChoice of
        miAlpha : AlphaOn := not AlphaOn;
        miBeta  : BetaOn := not BetaOn;
        miGamma : GammaOn := not GammaOn;
        miDelta : DeltaOn := not DeltaOn;
      end;
  until M.GetLastCommand = ccQuit;

  M.Erase;
  M.Done;
end.
```

This example is instructive in several ways, so let's go through all the differences from the initial version. First, we declare four boolean variables, AlphaOn and three more. These variables are incidental to this example; we just toggle them True or False when each associated item is selected. In a real program AlphaOn might correspond to WordWrapOn, BetaOn to AutoIndentOn, and so on. A real custom string proc can deal with any kind of status variables, not just booleans. All you need is to convert the status variable to a displayable string.

The next addition is CustomStringProc itself. It uses a case statement to distinguish among the various items. Some of the items aren't customized. For the others, we use the appropriate Boolean variable to merge an 'Off' or ' On' string into the Name. Note the use of the Move procedure to copy the string into the last three characters of the Name string.

This sample routine is pretty limited. If you'd like to see some real-world examples, take a look at the ones in MAKEMENU and MAKESCRN. These show how to merge not just booleans, but also numbers, enumerated types, characters, and other data types into the Name string.

The next changes occur within InitMenu. Both calls to AddFramedSubMenu have had their X-coordinates adjusted to leave space for the extra four characters (a separating space plus 'Off') needed for status.

The program's main block initializes the boolean status values, then enables the custom string procedure. In order to let you toggle status values repeatedly, it calls Process in a loop, until you press <Esc> at the top level menu. For each ccSelect command, it toggles the appropriate boolean value.

The Current Item Procedure

OPMENU has another hook that is useful for different reasons than the custom string procedure. Whereas the custom string procedure is called for every item, every time it is drawn, the current item procedure is called only when a different item is selected. This hook allows you to do things like write a detailed help line for each item as it is selected.

By default, OPMENU calls a routine that does nothing (NoCurrItemAction, interfaced by OPMENU). You can enable a routine of your own by calling Menu.SetCurrentItemProc:

```
procedure SetCurrentItemProc(IP : mnCurrentItemProc);
  {-Set procedure to be called whenever the current item is drawn}
```

The parameter passed to SetCurrentItemProc must be a procedure that is compiled under the FAR model (using the {$F+} compiler directive), is global, and matches the following prototype declaration:

```
{$F+}
procedure CurrentItemProc(CurrentItemPtr : MenuItemNodePtr; MPtr : MenuPtr);
```

Like many other user hooks in **Object Professional**, the current item procedure is called from a virtual method, CurrentItem. You can get the same effect by deriving a new type from Menu and overriding this method.

The current item procedure is passed a pointer to the item that was just made current, and a pointer to the menu system. It can access the methods and data fields of the referenced objects in order to perform its task. The current item procedure can perform any desired action, with the following restrictions: the active window must be the same on exit as it was on entry; the selected menu item cannot be changed.

Let's return once more to the introductory example and add one of these procedures. In this case, we'll just write a string to the last row of the screen showing which item key is current. Add the following global procedure to the example:

```
{$F+}
procedure CurrentItemProc(CurrentItemPtr : MenuItemNodePtr; MPtr : MenuPtr);
begin
  FastWrite(Long2Str(CurrentItemPtr^.ItemKey)+' is the current item key',
            ScreenHeight, 1, TextAttr);
end;
```

And activate this routine by adding the following to the main block of code:

```
{Enable the current item procedure}
M.SetCurrentItemProc(CurrentItemProc);
```

Of course, CurrentItemProc could be much fancier: it could write text within a bordered window; it could write several lines of text; and so on. Probably the most common use will be to display a dynamic context-sensitive help window for each menu item (like a Lotus-style menu does). MAKEMENU will generate this code for you if you enable its help option.

1. Low Level Objects

Each item in a menu system is managed by a MenuItemNode instance. You probably won't need to access the MenuItemNodes directly. The primary need for such direct access is for dynamic modification of menu systems: changing item name, position, or protection. MAKEMENU needs to do a lot of this, but a normal application program is unlikely to do so.

The other common need to access a MenuItemNode directly is to read its help string in order to feed it to a current item procedure. In this case, you can use the source code MAKEMENU generates without needing to understand the MenuItemNode object itself.

We'll describe only the methods of MenuItemNode that you're at all likely to need. The MenuItemList and SubMenu objects provide no methods that you are likely to need, since a) most of their methods are accessible via methods of the Menu object, or b) the methods override inherited methods with only subtle differences. Refer to OPMENU.PAS if you require additional information.

Constants

```
inProtected        = $01;
```

This bit is set in the MenuItemNode option byte to indicate that the item is protected.

```
DefItemOptions     : Byte = 0;
```

The default options for a MenuItemNode.

```
inAllocConsPtr     : Boolean = False;
```

When this constant is True, MenuItemNode.Load allocates space for an additional string, the "constant name" of the item. This string is read and written by MAKEMENU, and is used to generate user-selected constant names in source code. MAKEMENU sets it True; you shouldn't do so.

Types

```
MenuItemNodePtr    = ^MenuItemNode;
MenuItemNode       =
  object(DoubleListNode)
    ...
  end;
```

The MenuItemNode object.

```
MenuItemListPtr    = ^MenuItemList;
MenuItemList       =
  object(CircularList)
    ...
  end;
```

The MenuItemList object. No methods of this object are documented here. See OPMENU.PAS for a listing.

```
SubMenuPtr         = ^SubMenu;
SubMenu            =
  object(RawWindow)
    ...
  end;
```

The SubMenu object. No methods of this object are documented here. See OPMENU.PAS for a listing.

ChangeDisplayPos / ChangeHelp

Declaration

```
procedure ChangeDisplayPos(DisplayPos : Byte);
```

Purpose

Change the display offset.

Description

The display offset is the distance from the left/top edge of the MenuItemList's associated window to the start of the (padded) item, for horizontal/vertical lists, respectively. Position 1 is at the far left/top of the active window rectangle.

ChangeDisplayPos doesn't validate the specified position. If the position puts the item "out of order" relative to its sequence in the linked list of items, odd behavior will ensue.

Example

```
IPtr := M.FindItem(10);
IPtr^.ChangeDisplayPos(1);
```

Changes the display position of menu item 10 to the first row or column (for vertical or horizontal orientation, respectively).

See Also

GetDisplayPos Menu.AddItem

Declaration

```
function ChangeHelp(Help : String) : Boolean;
```

Purpose

Change the help string.

Description

This function works only if the help string was initially allocated on the heap (by calling AddItemHelp or by loading a menu system). Otherwise it returns False. If the help string is stored in a typed constant, modify the typed constant directly.

If the length of the new help string is different from the old, ChangeHelp deallocates the old heap block and allocates a new one whose length equals that of the Help parameter. If there is insufficient heap space, ChangeHelp exits, returning False. Otherwise it stores the new help string and returns True.

Example

```
IPtr := M.FindItem(10);
if not IPtr^.ChangeHelp('Copy files to a different directory') then
  {error};
```

Changes the help string associated with menu item 10.

See Also

HelpString Menu.AddItemHelp

Declaration
```
procedure ChangeKey(Key : LongInt);
```

Purpose
Change the item key.

Description
The item Key is a unique integer that identifies the MenuItemNode. ChangeKey doesn't validate the specified Key for uniqueness.

Example
```
M.ActiveItemPtr^.ChangeKey(100);
```
Changes the key of the current menu item to 100.

See Also
ItemKey Menu.AddItem

Declaration
```
function ChangeName(Name : String) : Boolean;
```

Purpose
Change the item name.

Description
This function works only if the item name was initially allocated on the heap (by calling AddItem, AddItemHelp, or by loading a menu system). Otherwise it returns False. If the name is stored in a typed constant, modify the typed constant directly.

If the length of the new name is different from the old, ChangeName deallocates the heap block used for the old name and allocates a new one. If there is insufficient heap space, ChangeName exits, returning False. Otherwise it stores the new name and returns True.

Example
```
if not M.ActiveItemPtr^.ChangeName('Copy') then {error};
```

See Also
DisplayInfo Menu.AddItem

Declaration
```
procedure ChangeSelectPos(SelectPos : Byte);
```

Purpose
Change the select character offset.

Description
The select character is drawn in a different attribute from the other characters in a menu item name. This character is also used to select menu items via alphanumeric searching. If SelectPos is zero, the selection feature is disabled. A SelectPos of 1 indicates the first character in the name (independent of padding). ChangeSelectPos does not validate that the specified SelectPos is within the length of the current name string.

Example
```
IPtr := M.FindItem(10);
IPtr^.ChangeSelectPos(2);
```
Changes the select position of menu item 10 to the second character of the name.

See Also
Menu.AddItem SelectChar

ChangeSubPtr / DisplayInfo

Declaration

```
procedure ChangeSubPtr(SPtr : SubMenuPtr);
```

Purpose

Change submenu pointer of item.

Description

Menu's AddSubMenu method calls ChangeSubPtr to link a newly created SubMenu into the current item. You could use it either to dynamically change the attached submenu or to have the same submenu attach to more than one item (hence saving memory space).

Examples

```
IPtr := M.FindItem(10);
IPtr^.ChangeSubPtr(OtherSubMenuPtr);
```

Changes the submenu associated with menu item 10 to the submenu pointed to by OtherSubMenuPtr.

```
var
  SM : SubMenuPtr;
...
with M do begin
  ...
  AddItem('First', 2, 1, miFirst);
    AddFramedSubMenu(3, 4, 9, 5, Vertical, Frame2);
    SM := ActiveSubPtr;
    AddItem('Alpha', 1, 1, miAlpha);
    AddItem('Beta', 2, 1, miBeta);
    ItemsDone;
  AddItem('Second', 9, 1, miSecond);
    ActiveItemPtr^.ChangeSubPtr(SM);
  ItemsDone;
end;
```

Saves the SubMenuPtr of the menu attached to 'First', then attaches the same submenu to 'Second'. Note that the position of the submenu is not adjusted automatically.

See Also

Menu.AddSubMenu

Declaration

```
procedure DisplayInfo(var DisplayPos, SelectPos : Byte; var S : String);
```

Purpose

Return string and display offsets for item.

Description

MenuItemList calls this method when it is drawing each item. It returns the display offset, selection character offset, and name string previously stored for the item.

Example

```
IPtr := M.FindItem(10);
IPtr^.DisplayInfo(DPos, SPos, Name);
```

Returns the display position, select position and name of the menu item with key 10.

See Also

GetDisplayPos HelpString

Declaration
```
function GetDisplayPos : Byte;
```

Purpose
Return just the display offset.

Description
The display offset is the distance from the left/top edge of the MenuItemList's associated window to the start of the (padded) item, for horizontal/vertical lists, respectively. Position 1 is at the far left/top of the active window rectangle.

Example
```
IPtr := M.FindItem(10);
DPos := IPtr^.GetDisplayPos;
```

See Also
ChangeDisplayPos DisplayInfo

Declaration
```
function HelpString : String;
```

Purpose
Return help string for item.

Description
If no help string has been stored, HelpString returns the empty string. Otherwise it returns the help string saved by AddItemHelp, AddItemHelpPtr, or ChangeHelp.

Example
```
{$F+}
procedure MyItemAction(CurrentItem : MenuItemNodePtr; MPtr : MenuPtr);
begin
  FastWrite(Pad(CurrentItem^.HelpString, ScreenWidth), ScreenHeight, 1,
    TextAttr);
end;
```
This is a help procedure (activated by SetCurrentItemProc) that uses the HelpString method to display the help string associated with the current menu item.

See Also
ChangeHelp Menu.AddItemHelp

Declaration
```
function IsProtected : Boolean;
```

Purpose
Return True if item is protected.

Description
Newly created items are not protected. Change protection by calling Protect, Unprotect, or (at a higher level) ProtectItem or UnprotectItem.

Example
```
IPtr := M.FindItem(10);
if IPtr^.IsProtected then {do something};
```
Takes some action if the item 10 is protected.

See Also
Menu.ProtectItem Protect

ItemKey / Protect / SelectChar

Declaration
```
function ItemKey : LongInt;
```

Purpose
Return menu key for item.

Description
The item key is a unique integer value identifying the MenuItemNode. It is assigned by AddItem or ChangeKey. After a call to Process, the chosen item key is returned by MenuChoice.

Example
```
Key := M.ActiveItemPtr^.ItemKey.
```
Returns the key of the current menu item.

See Also
ChangeKey
Menu.MenuChoice
Menu.AddItem

Declaration
```
procedure Protect;
```

Purpose
Protect item.

Description
Protected items are displayed in a different color and cannot be selected. A higher level method of protecting an item is Menu.ProtectItem. Note that it is a programming error to protect the current item using Protect. Menu.ProtectItem avoids this error by automatically adjusting the current item if it becomes protected.

Example
```
IPtr := M.FindItem(10);
IPtr^.Protect;
```
Protects item 10.

See Also
Menu.ProtectItem
Unprotect

Declaration
```
function SelectChar : Char;
```

Purpose
Return alpha selection char (uppercase).

Description
If the SelectPos is zero or greater than the length of the item name, SelectChar returns #0. Otherwise it returns the uppercase value of the character at the specified position of the name string. MenuItemList calls this method to search for an alpha selection.

Example
```
IPtr := M.FindItem(10);
C := IPtr^.SelectChar;
```
Returns the uppercase select character of menu item 10.

See Also
ChangeSelectPos
Menu.AddItem

Declaration
```
function SubPtr : SubMenuPtr;
```

Purpose
Return submenu pointer for item.

Description
If the item has no submenu, SubPtr returns nil. Otherwise it returns the address of the attached SubMenu object.

Example
```
IPtr := M.FindItem(10);
SPtr := IPtr^.SubPtr;
```
Returns the submenu pointer of the menu item with a key of 10.

See Also
ChangeSubPtr

Declaration
```
procedure Unprotect;
```

Purpose
Unprotect item.

Description
Protected items are displayed in a different color and cannot be selected. Newly created items are unprotected by default. A higher level method of unprotecting an item is Menu.UnprotectItem.

Example
```
IPtr := M.FindItem(10);
IPtr^.Protect;
```

See Also
IsProtected
Protect
Menu.UnprotectItem

2. Menu: Menu Systems

This section describes the constants, types, variables, and methods of the Menu object. See the introduction above for more information about menu systems.

The following Menu methods do not appear in the detailed descriptions following. These methods work just like their counterparts in the Frame and RawWindow objects, except that here they automatically apply to the main frame of the current window in the menu system. They may be called while the menu is being initialized (see "Menu Customization" in the introduction).

```
procedure AddHeader(S : string; Posn : HeaderPosType);
  {-Add a standard header}
procedure AddHeaderColor(S : string; Posn : HeaderPosType;
                         AttrColor, AttrMono : Byte);
  {-Add a standard header with custom color}
procedure AddCustomHeader(S : string; Posn : FrameCornerType;
                          DX, DY : Integer; AttrColor, AttrMono : Byte);
  {-Add a custom header}
{$IFDEF UseShadows}
procedure AddShadow(Posn : ShadowPosType; DrawType : ShadowDrawType);
  {-Add a standard shadow}
procedure AddShadowColor(Posn : ShadowPosType; DrawType : ShadowDrawType;
                         AttrColor, AttrMono : Byte);
  {-Add a standard shadow with custom color}
{$ENDIF}
procedure EnableExplosions(StageDelay : Word);
  {-Enable exploding windows with StageDelay milliseconds per stage}
procedure EnableNormalOpen;
  {-Disable exploding windows or alternate draw routines}
```

Constants

```
mnAlphaMatch      = $0001;      {Check for alpha matches}
mnSelectOnMatch   = $0002;      {Implicit <Enter> on alpha char match}
mnAllowPending    = $0004;      {Submenu stays down after passing no submenu}
mnArrowSelect     = $0008;      {Select submenus with arrows}
mnPopOnSelect     = $0010;      {Pop current submenu after selecting an item}
mnAllHotSpots     = $0020;      {Exit on any mouse hotspot in menu system}
mnUseItemForTopic = $0040;      {Use the current item key for help topic,
                                 else use the code in the command window}
mnSelectOnClick   = $0080;      {Implicit <Enter> on first mouse click}
mnMouseSupport    = $0100;      {Set when mouse support is requested
                                 (used by MAKEMENU only)}
mnMainSelect      = $0200;      {Set when main menu shows selection}
mnMainHighlight   = $0400;      {Set when main menu shows highlight}
```

These are the options that affect the behavior of a Menu. If mnAlphaMatch is set, Menu's Process method will search the current item list for an item whose SelectChar method result matches any alphanumeric character entered. If mnSelectOnMatch is enabled, the matching item will be selected as well.

When mnAllowPending is enabled, Process "remembers" that a submenu was pulled down when an arrow key movement causes it to temporarily select an item with no submenu. The simplest example of this behavior is embodied by a three-item top level menu, where the left and right items have submenus and the center item has none. If the leftmost submenu is pulled down and the right arrow key is pressed, the submenu will be erased and selection bar will move to the center item. If the right arrow key is pressed again, the selection bar will move to the right item and its submenu will automatically appear.

mnArrowSelect determines whether the arrow keys will have an effect when a key is pressed whose direction is perpendicular to the orientation of the current item list. Suppose that the current item list is vertical and the right arrow key is pressed. If mnArrowSelect is enabled, then Process first checks to see if the parent list of the current submenu is horizontally oriented. If so, and if the parent has additional unprotected items, the current submenu will be erased and the next item in the parent menu will be selected. If the parent's orientation is not horizontal, Process will attempt to select a submenu of the current item. If mnArrowSelect is not enabled, arrow keys produce action only when their direction is parallel to the current list orientation.

When mnPopOnSelect is enabled, Process automatically erases the current submenu whenever the user makes a selection (exiting with ccSelect). This option will not cause Process to erase the top level menu, however.

mnAllHotSpots controls the response of the Process method to mouse clicks over hot spots. When mnAllHotSpots is disabled, Process reacts to hot spots defined for the current submenu only. When it is enabled, Process reacts to hot spots in all active submenus. When Process does react, it exits with the ccMouseSel command code. Note that, by default, no submenus will have hot spots.

If mnUseItemForTopic is set, Process passes the low word of the current item key as the topic number to the user help routine. Otherwise, the help topic is the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

mnSelectOnClick determines whether a mouse click causes immediate selection of a menu item, or whether a second click over the same item will be required to select it.

mnMouseSupport is used by MAKEMENU to flag a menu stream that has requested mouse support. The OPMENU unit ignores this option.

mnMainSelect controls whether Menu.Draw displays the selection bar on the top level menu. mnMainHighlight controls whether Menu.Draw highlights the alpha search characters on the top level menu. Menu constructors, Process, SelectItem, and SelectSubMenu force both of these options on. EraseAllSubMenus forces them on or off depending on the parameters passed to it.

See the introductory discussion of "Menu Commands" for further information about many of these options.

```
DefMenuOptions     : Word = mnAlphaMatch+mnSelectOnMatch+mnArrowSelect+
                            mnAllHotSpots;
```

The default options for a Menu. Also note the comments about mnMainSelect and mnMainHighlight above.

```
BadMenuOptions     : Word = mnSubPending;
```

Options used internally by Menu. The mnOptionsOn and mnOptionsOff methods will not change these bits of a Menu option word.

```
MenuKeyMax = 110;
MenuKeyID  : string[16] = 'opmenu key array';
MenuKeySet : array[0..MenuKeyMax] of Byte = (...);
MenuCfgEnd : Byte = 0;
```

MenuKeyId is an ID string used to mark the beginning of the configuration data area for OPMENU; MenuCfgEnd marks the end of the configuration data area. In between them is MenuKeySet, the command table used by MenuCommands. MenuKeyMax is the last valid index into the table.

```
MenuDepth          : Word = 8;
```

The initial size of the PointerStack objects allocated by Menu to manage nested submenus. If more levels than this are activated simultaneously, the PointerStacks will automatically resize themselves (growing by a factor of two) but will fragment the heap. The Menu object consumes 8 bytes of heap space for each increment in MenuDepth.

```
mnNotAKey          = $FFFFFFFF;
```

This item key value, which is equivalent to -1, is reserved by OPMENU to mean "no particular key." If you pass this value to ItemsDoneCustom, OPMENU will select the first item on the submenu as the default. If you call MenuChoice before calling Process, it will return mnNotAKey.

Types

```
mnCustomProc       = procedure (var Name : String; Key : LongInt;
                                Selected, Highlighted : Boolean;
                                WPtr : RawWindowPtr);
```

Procedural type that defines the parameters of a user-supplied custom string procedure. See the discussion of "The Custom String Procedure" above for more information.

```
mnCurrentItemProc = procedure (CurrentItem : MenuItemNodePtr; MPtr : MenuPtr);
```

Procedural type that defines the parameters of a user-supplied current item procedure. See the discussion of "The Current Item Procedure" above for more information.

```
mnOrientation      = (Vertical, Horizontal, NoOrient);
```

Possible orientations for a MenuItemList. The value NoOrient indicates an improperly initialized list.

```
mnItemProc         = procedure (IPtr : MenuItemNodePtr; var D; MPtr : MenuPtr);
mnWindowProc       = procedure (WPtr : RawWindowPtr; var D; MPtr : MenuPtr);
mnListProc         = procedure (LPtr : MenuItemListPtr; var D; MPtr : MenuPtr);
```

Procedural types for routines called by the VisitAllItems, VisitAllWindows, and VisitAllItemLists methods, respectively. The first parameter gives the current item, window, or item list, respectively. The second, untyped parameter is available to pass user-defined data to each routine. The third parameter is always the address of the menu system.

```
MenuPtr = ^Menu;
Menu =
  object(CommandWindow)
    ...
  end;
```

The Menu object.

Variables

```
MenuCommands : CommandProcessor;
```

The default CommandProcessor for a Menu. This variable is initialized by OPMENU's initialization block.

Declaration
```
function ActiveItemPtr : MenuItemNodePtr;
```

Purpose
Return pointer to currently active item.

Description
If the menu system is being initialized (calls to ItemsDone are still required), ActiveItemPtr returns the address of the item most recently added to the menu system. If the menu system is fully initialized but not on-screen, ActiveItemPtr returns the address of the top level menu's current item. Otherwise it returns the address of the active submenu's current item.

Example
```
HelpStr := M.ActiveItemPtr^.HelpString;
```
Stores the help string of the active item.

See Also
ActiveWinPtr

Declaration
```
function ActiveListPtr : MenuItemListPtr;
```

Purpose
Return pointer to active item list.

Description
If the menu system is being initialized (calls to ItemsDone are still required), ActiveListPtr returns the address of the item list most recently added to the menu system. (This is the main item list if no submenus have been added yet, or the list associated with the submenu to which items are currently being added.) If the menu system is fully initialized but not on-screen, ActiveListPtr returns the address of the top level menu's list. Otherwise it returns the address of the active submenu's list.

Example
```
if ActiveListPtr^.Orientation = Vertical then
  {do something} ;
```
Detects whether the active item list is vertical or horizontal.

See Also
ActiveWinPtr

Declaration
```
function ActiveSubPtr : SubMenuPtr;
```

Purpose
Return pointer to active submenu.

Description
If no submenu is active, ActiveSubPtr returns nil. Otherwise it returns the address of the SubMenu most recently displayed or added with AddSubMenu.

See Also
MenuItemNode.ChangeSubPtr

ActiveWinPtr / AddFramedSubMenu

Declaration

```
function ActiveWinPtr : RawWindowPtr;
```

Purpose

Return pointer to active RawWindow.

Description

If the menu system is being initialized (calls to ItemsDone are still required), ActiveWinPtr returns the address of the submenu most recently added to the menu system. (This is the Menu itself if no submenus have been added yet.) If the menu system is fully initialized but not on-screen, ActiveWinPtr also returns the address of the menu itself. Otherwise it returns the address of the active submenu. Note that SubMenu and Menu are derived directly and indirectly from RawWindow; hence, both can be treated as a RawWindow and all the methods and data of that object are available for access.

Example

```
M.ActiveWinPtr^.ChangeHeader(HdrNum, NewHdrStr);
```

Changes a header associated with the active window. Note that any other RawWindow (or Frame) method can be accessed similarly.

See Also

ActiveItemPtr

Declaration

```
procedure AddFramedSubMenu(X1, Y1, X2, Y2 : Byte;
                           Orient : mnOrientation;
                           FA : FrameArray);
```

Purpose

Add a framed submenu with options and colors cloned from main menu.

Description

A framed submenu is attached to the current item. X1, Y1, X2, and Y2 specify the active coordinates of the submenu, not counting the border. Orient is Vertical or Horizontal, specifying the orientation of the item list. FA specifies an array of border characters. The window options of the submenu's RawWindow will match those of the top level menu. The colors used by the SubMenu and its MenuItemList will match those of the top level menu.

If a Menu error is pending on entry to AddFramedSubMenu, it exits immediately.

See "Building a Menu System" for more information and a complete example program.

Example

```
M.AddFramedSubMenu(4, 4, 20, 10, Vertical, DefWindowFrame);
```

Adds a framed submenu to the current menu item (the last menu item added). The frame will be one row and one column outside the submenu coordinates: absolute screen location 3,3 for the top-left frame corner; absolute screen location 21,11 for the lower-right frame corner. The items in this submenu will be arranged vertically.

See Also

AddSubMenu AddSubMenuCustom

Declaration

```
procedure AddItem(Name : String; DisplayPos, SelectPos : Byte; Key : LongInt);
```

Purpose

Add item to current item list, allocating string space on heap.

Description

An item of the specified Name and Key is added to the end of the current item list. The new item becomes the current item. DisplayPos specifies the item's coordinate relative to the left or top edge of the window, depending on whether the submenu has horizontal or vertical orientation. SelectPos specifies which character of Name to highlight for alphanumeric selection. If SelectPos is zero, this feature is disabled. Key is a LongInt value that uniquely identifies this menu item.

A copy of the Name string is stored on the heap. If there is insufficient heap space, AddItem generates an error (epFatal+ecOutOfMemory).

If a Menu error is pending on entry to AddItem, it exits immediately.

See "Building a Menu System" for more information and a complete example program.

Examples

```
M.AddItem('File', 4, 1, 1);
```

Adds an item named 'File' to the current menu level. Assuming the current menu uses horizontal orientation, the string 'File' will be displayed four columns from the left edge of the window. The first position of the string ('F'), will be the select character. The item key is 1.

```
M.AddFramedSubMenu(4, 4, 20, 10, Vertical, DefWindowFrame);
  M.AddItem('New', 1, 1, 2);
  M.AddItem('Save', 2, 1, 3);
  M.ItemsDone;
```

Creates a submenu with two items: 'New' and 'Save'. 'New' will be on the first (top) row of the window and will use the first character ('N') as the select character. 'Save' will be on the second row of the window and also uses its first character as the select character. The items will have keys of 2 and 3, respectively.

See Also

AddItemPtr

Declaration

```
procedure AddItemHelp(Name : String; DisplayPos, SelectPos : Byte;
                      Key : LongInt; Help : String);
```

Purpose

Add item to current item list, allocating Name and Help strings on heap.

Description

Works just like AddItem, except that a copy of the Help string is also stored on the heap. This help string can be accessed with MenuItemNode.HelpString from within a current item procedure. OPMENU does not do anything with the help string except to store and retrieve it for you.

Example

```
M.AddItemHelp('File', 4, 1, 1, 'Process files');
```

Adds an item named 'File' to the current menu level. The help string 'Process files' will be stored on the heap and can be recalled by a help display routine.

See Also

AddItemHelpPtr SetCurrentItemProc

AddItemHelpPtr

Declaration

```
procedure AddItemHelpPtr(var Name : String; DisplayPos, SelectPos : Byte;
                         Key : LongInt; var Help : String);
```

Purpose

Add item to current item list, not allocating Name or Help strings on heap.

Description

Works just like AddItemPtr, except that the address of the Help string is also stored. This help string can be accessed with MenuItemNode.HelpString from within a current item procedure. OPMENU does not do anything with the help string except to store and retrieve it for you.

Examples

```
const
  FileStr : String[4] = 'File';
  FileHelpStr : String[13] = 'Process files';
begin
  ...
  M.AddItemHelpPtr(FileStr, 4, 1, 1, FileHelpStr);
```

Adds an item named 'File' to the current menu level. The contents of the string FileHelpStr will be returned by the method HelpString.

```
var
  ItemStr : String;
  ItemHelpStr : String;
begin
  ...
  ItemStr := 'File';
  ItemHelpStr := 'Process files';
  M.AddItemHelpPtr(ItemStr, 4, 1, 1, ItemHelpStr);
  ItemStr := 'Save';
  ItemHelpStr := 'Save the current file';
  M.AddItemHelpPtr(ItemStr, 5, 1, 2, ItemHelpStr);
```

Incorrect! AddItemHelpPtr saves the address of the variables ItemStr and ItemHelpStr, not their values. If you change their values, you change the menu display. This example will display 'Save' where you expect 'File'. In general, you should use only typed constants as arguments to AddItemHelpPtr.

See Also

AddItemHelp AddItemPtr

Declaration

```
procedure AddItemPtr(var Name : String; DisplayPos, SelectPos : Byte;
                     Key : LongInt);
```

Purpose

Add item to current item list, not allocating Name string on heap.

Description

Works like AddItem, except that AddItemPtr does not allocate heap space for the Name string. It instead assumes that the variable passed in the Name position retains its value throughout the life of the menu system. Typically this means that a typed constant should be passed for Name. AddItemPtr is more efficient than AddItem because it avoids making duplicate copies of each item Name. However, it is your responsibility to assure that the name strings remain valid while the menu system is used. Note that MAKEMENU will generate the appropriate typed constants and calls to AddItemPtr if you specify the "string constants" option.

Be aware that Turbo Pascal typed constants are stored in the data segment. An advantage of storage in the data segment is that the constants are never overlaid by Turbo's OVERLAY unit. On the other hand, the data segment is limited to 64K bytes, and a large menu system may use a significant portion of that space.

Examples

```
function InitMenu(var M : Menu) : Word;
const
  FileStr : String[4] = 'File';
  EditStr : String[4] = 'Edit';
begin
  ...
  M.AddItemPtr(FileStr, 4, 1, 1);
  M.AddItemPtr(EditStr, 20, 1, 4);
```

Adds items named 'File' and 'Edit' to the current menu level. Note that the typed constants may be declared within a procedure. Turbo Pascal typed constants retain their values even when the scope where they are declared is not active.

```
var
  ItemStr : String;
begin
  ...
  ItemStr := 'File';
  M.AddItemPtr(ItemStr, 4, 1, 1);
  ItemStr := 'Save';
  M.AddItemPtr(ItemStr, 5, 1, 2);
```

Incorrect! AddItemPtr saves the address of the variable ItemStr, not its value. If you change the value, you change the menu display. This example will display 'Save' where you expect 'File'.

See Also

AddItemHelpPtr

AddSeparator / AddSeparatorColor

Declaration

```
procedure AddSeparator(FirstChar, SpanChar, LastChar : Char; DisplayPos : Byte);
```

Purpose

Add a separator bar at offset DisplayPos within the window.

Description

AddSeparator generates an appropriate call to AddSpanHeaderColor for the main frame of the active window. If the associated item list is vertical, the span bar is horizontal, and vice versa. FirstChar is the character that joins the window's border at the top or left edge; LastChar joins at the bottom or right edge. SpanChar is duplicated to span the window. For a borderless menu, specify that FirstChar and LastChar are the same as SpanChar. DisplayPos is the offset from the left or top edge (for horizontal or vertical menus, respectively) to where the separator joins the frame. Note that DisplayPos is defined consistently with the same parameter passed to AddItem. For example, if you had items on rows 1, 2, and 4 of a vertical menu and you wanted a separator bar between them you would specify DisplayPos = 3 to AddSeparator.

The color of the separator bar matches that of the border characters for the window.

If a Menu error is pending on entry to AddSeparator, it exits immediately.

See "Menu Customization" for more information and a complete example program.

Example

```
M.AddSeparator(#195, #196, #180, 3);
```

Adds a separator bar to the current menu level, using the single-line character set, on the third row (assuming this is a vertical menu) from the top.

Declaration

```
procedure AddSeparatorColor(FirstChar, SpanChar, LastChar : Char;
                            DisplayPos : Byte; AttrColor, AttrMono : Byte);
```

Purpose

Add a separator bar at offset DisplayPos within the window with specified colors.

Description

Works just like AddSeparator, except that the separator bar will be drawn using the specified attributes.

Example

```
M.AddSeparatorColor(#195, #196, #180, 3, $02, $07);
```

Adds a color separator bar to the current menu level (green on black for color monitors, normal video for monochrome monitors).

Declaration

```
{$IFDEF UseShadows}
procedure AddShadows(MainMenu : Boolean; Posn : ShadowPosType;
                     DrawType : ShadowDrawType);
```

Purpose

Add shadows to all submenus.

Description

If MainMenu is False, shadows will be added to all submenus but not to the top level menu. Posn specifies the shadow position (shBR or shBL) and DrawType specifies the drawing style (shSeeThru or shOverWrite). The shadow color is determined by the values of ShadowColor and ShadowMono in the ColorSet used to initialize the menu system.

This method should be called only after all submenus and items have been added to the menu system. Any existing shadows are disposed of before adding the new ones.

If a Menu error is pending on entry to AddShadows, it exits immediately.

Because adding each shadow requires allocation of heap space, this method may generate an out of memory error.

Example

```
M.AddShadows(False, shBR, shOverWrite);
```

Adds opaque, bottom-right shadows to all submenus (but not the main menu).

See Also

Frame.AddShadow

Declaration

```
procedure AddSubMenu(X1, Y1, X2, Y2 : Byte; Orient : mnOrientation);
```

Purpose

Add an unframed submenu with options and colors cloned from main menu.

Description

Works just like AddFramedSubMenu, except that the new submenu is borderless.

When defining overlapping submenus for a Lotus-style menu system, be sure to specify the same values for X1 and X2 for all overlapping levels. When you do so, OPMENU knows to overwrite and restore the entire region shared by the menu bar.

Example

```
M.AddSubMenu(1, 1, ScreenWidth, 1, Horizontal);
```

Adds an unframed submenu to the current menu item (the last menu item added). This submenu is a single row high and spans the top row of the screen. Items will be arranged horizontally. These parameters are the hallmark of a Lotus-style menu.

See Also

AddFramedSubMenu AddSubMenuCustom

AddSubMenuCustom / CurrentItem

Declaration

```
procedure AddSubMenuCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                           Options : LongInt; Orient : mnOrientation;
                           FA : FrameArray);
```

Purpose

Add a submenu with custom window options.

Description

This method provides all the flexibility of AddSubMenu and AddFramedSubMenu, plus additional control of colors and window options.

The following fields from the ColorSet will apply to the submenu's item list:

```
TextColor, TextMono          - normal items
SelItemColor, SelItemMono    - selected item
HighItemColor, HighItemMono  - highlight characters
ProItemColor, ProItemMono    - protected items
```

The fields normally associated with the window border, headers and shadows will apply to the SubMenu itself.

Before using the specified window Options, AddSubMenuCustom forces the wResizeable option off, and the wUserContents and wClear options on.

Example

```
M.AddSubMenuCustom(4, 4, 20, 10, MyColorSet,
                   DefWindowOptions or wBordered or wSoundEffects,
                   Vertical, DefWindowFrame);
M.ActiveWinPtr^.EnableExplosions(15);
```

Adds a framed submenu to the current menu item (the last menu item added). The border will be drawn using the default window frame characters. This submenu will use the specified color set and will produce sound effects during explosions.

See Also

AddFramedSubMenu AddSubMenu

Declaration

```
procedure CurrentItem(IPtr : MenuItemNodePtr); virtual;
```

Purpose

Called just after the currently selected item is drawn.

Description

This method calls the "current item procedure" that was specified by SetCurrentItemProc. Types derived from Menu can override this method to provide alternate behavior.

See the discussion of "The Current Item Procedure" in the introduction to this section.

See Also

SetCurrentItemProc

Declaration
```
procedure DefaultItem(Key : LongInt);
```

Purpose
Make item Key the default of its submenu. No screen update.

Description
If the specified item Key is found anywhere in the menu system, DefaultItem makes the associated item the default of its submenu. The next time that submenu is selected, the item will become the current item. DefaultItem doesn't update the screen, whether the menu system is already visible or not.

Examples
```
M.DefaultItem(11);
```
Makes the menu item with a key of 11 the default item in its submenu.

```
M.DefaultItem(11);
M.DefaultItem(10);
```
Assuming items 10 and 11 are in the same submenu, item 10 will be the default item, since it was the last default set.

See Also
SelectItem

Declaration
```
procedure DefaultPath(Key : LongInt);
```

Purpose
Select path to specified item.

Description
If the menu system is already on-screen, DefaultPath calls SelectItem to select the specified item and update the screen. Otherwise, DefaultPath finds the specified item and builds an internal stack. The next time Process or Draw is called, all submenus leading to the item will be displayed, and the item will be highlighted.

If no item matching Key is found, DefaultPath does nothing.

Example
```
repeat
  M.Process;
  case M.GetLastCommand of
    ccQuit : Finished := True;
    ccUser0 : M.DefaultPath(11);
  else
    case M.MenuChoice of
      1 : {Execute choice 1};
      2 : {Execute choice 2};
      {...}
    end;
  end;
until Finished;
```
The exit command ccUser0 is used as a short-cut to menu item 11. The next time around the loop, item 11 will be highlighted, ready for selection.

See Also
SelectItem

Done / EnableAllAlternateOpen

Declaration

```
destructor Done; virtual;
```

Purpose

Destroy a menu system.

Description

All heap space allocated by the menu system is released. Done uses a recursive algorithm to trace all items and all submenus. If the menu system is visible at the time Done is called, it is erased first.

Example

```
M.Done;
```

Erases the menu system from the screen and deallocates all heap space it uses.

See Also

CommandWindow.Done

Declaration

```
procedure EnableAllAlternateOpen(OpenProc, CloseProc : RawWindowProc;
                                 StageDelay : Word);
```

Purpose

Enable alternate draw and erase routines.

Description

EnableAllAlternateOpen calls the RawWindow's EnableAlternateOpen method for every window in the menu system.

This method should be called only after all submenus and items have been added to the menu system.

If a Menu error is pending on entry to EnableAllAlternateOpen, it exits immediately.

Example

```
M.EnableAllAlternateOpen(MyOpenProc, MyCloseProc, 10);
```

Turns on the use of the alternate open/close procedures (MyOpenProc, MyCloseProc) using a 10 millisecond delay between stages.

See RawWindow.EnableAlternateOpen for a complete example of alternate open/close procedures.

See Also

EnableAllExplosions

EnableAllNormalOpen

Declaration

```
procedure EnableAllExplosions(StageDelay : Word);
```

Purpose

Enable exploding windows with StageDelay milliseconds per stage.

Description

EnableAllExplosions calls the RawWindow's EnableExplosions method for every window in the menu system.

This method should be called only after all submenus and items have been added to the menu system.

If a Menu error is pending on entry to EnableAllExplosions, it exits immediately.

Example

```
M.EnableAllExplosions(10);
```

Turns on exploding/imploding windows in the main menu and all submenus.

See Also

EnableAllAlternateOpen

Declaration

```
procedure EnableAllNormalOpen;
```

Purpose

Disable exploding windows or alternate draw routines.

Description

EnableAllNormalOpen calls the RawWindow's EnableNormalOpen method for every window in the menu system.

This method should be called only after all submenus and items have been added to the menu system.

If a Menu error is pending on entry to EnableAllNormalOpen, it exits immediately.

Example

```
M.EnableAllNormalOpen;
```

Turns off exploding/imploding windows in the main menu and all submenus.

See Also

EnableAllAlternateOpen

Erase

Declaration

```
procedure Erase; virtual;
```

Purpose

Erase menu system. Remembers previous submenus for next draw.

Description

Overrides the normal Erase method in order to erase all visible submenus and the top level menu. The object keeps an internal record of what submenus were visible; the next time Draw or Process is called, the same submenus will be drawn and the same items highlighted.

Like StackWindow.Erase, this method should be called only if the menu system contains the current window. That is, if another non-menu window has been drawn on top of the menu system, Erase won't work. To deal with this situation, you must call StackWindow.EraseHidden instead. (EraseHidden calls Erase after switching to a virtual screen and erasing all the windows on top of the specified window. Because Erase is virtual, the call to erase the menu is directed to the method described here.)

Example

```
M.Erase;
```

Erases the menu but does not reset any of its initial values. The next time the menu is drawn it will be in exactly the state it was before it was erased.

See Also

EraseAllSubMenus
EraseClear
StackWindow.Erase

Declaration

```
procedure EraseAllSubMenus(ShowSelect, ShowHighlight : Boolean);
```

Purpose

Erase all submenus, leave main menu showing in specified state.

Description

Erases all submenus but not the top level menu. The top level's item list is redrawn consistent with the parameters. ShowSelect controls whether the current item has a selection bar. ShowHighlight controls whether the alpha search characters in each item are highlighted. EraseAllSubMenus updates the mnMainSelect and mnMainHighlight option flags as well.

If the menu system is not current, EraseAllSubMenus exits immediately.

Example

```
repeat
  M.Process;
  case M.GetLastCommand of
    ccQuit :
      begin
        M.EraseAllSubMenus(False, False);
        Finished := True;
      end;
    ccSelect :
      case M.MenuChoice of
        1..8 :
          {...Execute menu choices...} ;
        9..11 :
          begin
            {...Execute menu choices...}
            M.EraseAllSubMenus(True, True);
          end;
        {...}
      end;
  end;
until Finished;
```

If <Esc> is pressed while the top level menu is active (producing the ccQuit exit command) the top level menu is left visible but with the highlights removed, and the processing loop exits. After selecting menu choices 1 through 8 the menu system is left unchanged. For menu options 9, 10 and 11, we erase all submenus but continue processing at the top level.

See Also

Erase
EraseClear
EraseCurrentSubMenu

EraseClear / EraseCurrentSubMenu

Declaration

```
procedure EraseClear;
```

Purpose

Erase menu and clear active submenus for next draw.

Description

Works like Erase, except that the next time the menu system is drawn, only the top level menu will appear, no matter how many submenus were nested at the time it was erased. The default item in each submenu retains its previous value, however.

Example

```
M.EraseClear;
```

Removes all menus from the screen and clears the internal menu stack.

See Also

Erase EraseAllSubMenus

Declaration

```
procedure EraseCurrentSubMenu;
```

Purpose

Erase current submenu.

Description

Just the current submenu is erased. This method has no effect if the top level menu is the only visible menu at the time of the call. The selection and highlight characteristics of the top level menu are not changed even if it is the only menu left visible after the call (in contrast to the EraseAllSubMenus method).

Note that the same effect can be obtained automatically by setting the mnPopOnSelect option. However, the option works non-selectively, whereas EraseCurrentSubMenu can be applied to specified items, as shown in the example following.

Example

```
repeat
  M.Process;
  case M.GetLastCommand of
    ccQuit : begin
               M.EraseAllSubMenus(False, False);
               Finished := True;
             end;
    ccSelect :
      case M.MenuChoice of
        1..8 : {...Execute menu choices...} ;
        9..11 : begin
                  {...Execute menu choices...}
                  M.EraseCurrentSubMenu;
                end;
        {...}
      end;
  end;
until Finished;
```

After selecting menu choices 1 through 8 the menu system is left unchanged. For menu options 9, 10 and 11, we erase the current submenu but continue processing at the previous level.

See Also

EraseAllSubMenus

Declaration
```
function FindItem(Key : LongInt) : MenuItemNodePtr;
```

Purpose
Return pointer to item node.

Description
FindItem searchs the entire menu tree for the item with the specified Key. If not found, it returns nil. Otherwise, it returns the address of the associated MenuItemNode, which may be used to access the methods and data fields of the item's object.

Example
```
case M.GetLastCommand of
  ccQuit  : Finished := True;
  ccUser0 : begin
              IPtr := M.FindItem(10);
              if not IPtr^.ChangeName('Copy') then
                {error};
              M.Redraw;
            end;
  ccSelect :
    {Process menu choice} ;
end;
```
User exit command 0 changes the name of menu item 10 to 'Copy'.

See Also
ActiveItemPtr

Declaration
```
procedure GetHelpAttr(var Color, Mono : Byte);
```

Purpose
Get the colors stored for the help line.

Description
This routine simply returns the attributes stored by SetHelpAttr. OPMENU never actually uses these colors; it stores them simply for the convenience of user-written current item routines.

Example
```
{$F+}
procedure CurrentItemProc(CurrentItem : MenuItemNodePtr; MPtr : MenuPtr);
var
  Color, Mono, Attr : Byte;
begin
  MPtr^.GetHelpAttr(Color, Mono);
  if UseColor then
    Attr := Color
  else
    Attr := Mono;
  FastWrite(CurrentItem^.HelpString, ScreenHeight, 1, Attr);
end;
```
A current item routine that writes the item's help string to the bottom row using the video attributes stored by SetHelpAttr.

See Also
SetCurrentItemProc SetHelpAttr

Init

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; Orient : mnOrientation);
```

Purpose

Create a menu with default options and colors.

Description

This constructor instantiates a Menu using the default window options (DefWindowOptions), the default color set (DefaultColorSet), and the default menu options (DefMenuOptions). See InitCustom for more details.

Example

```
if not M.Init(1, 1, ScreenWidth, 1, Horizontal) then begin
  WriteLn('Failed to Init Menu,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates a horizontal menu using default window and color options. Because the menu's active coordinates span the entire screen we can infer that this is a borderless menu.

See Also

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; Orient : mnOrientation);
```

Purpose

Create a menu with custom window options.

Description

This constructor specifies the characteristics of the top level menu of a menu system. Items and submenus are added by later calls to AddItem, AddSubMenu, and related methods.

X1, Y1, X2, and Y2 specify the active window coordinates within which the item list will appear.

InitCustom plucks certain fields out of the specified ColorSet to use while drawing the menu. Besides those used by windows in general, InitCustom stores the following fields within the MenuItemList associated with the top level menu:

```
TextColor, TextMono            - normal items
SelItemColor, SelItemMono      - selected item
HighItemColor, HighItemMono    - highlight characters
ProItemColor, ProItemMono      - protected items
```

Note that the highlight characters apply only to unselected, unprotected items. The selected item is drawn uniformly in SelItemColor or SelItemMono.

Options are the bit-mapped flags that apply to all RawWindow-derived objects, such as wBordered. See the RawWindow Constants section for more details. InitCustom forces the wUserContents flag on, since Menu provides an UpdateContents method that can regenerate the window's contents on request. It forces the wResizeable flag off, since menus cannot be resized (except by MAKEMENU, which takes special precautions.)

Orient is the orientation of the top level menu's item list. It can be either Horizontal or Vertical.

Besides the heap space allocated for a CommandWindow, InitCustom also allocates two PointerStack instances, each capable of holding MenuDepth (defaults to 8) pointers. These stacks are used to manage the active submenus. If the menu system will be deeper than 8 levels, you should adjust MenuDepth (which is a typed constant) before calling the constructor. This will avoid the heap fragmentation that would occur when the PointerStacks resized themselves.

InitCustom assigns the default menu options (DefMenuOptions) to the menu, but it forces the mnMainSelect and mnMainHighlight options on.

The MenuCommands command processor is assigned to the Menu. The menu is assigned a unit code of ucMenu (for the purposes of identifying it to error and help handlers). The hardware cursor is set to a default mode of cuHidden.

The user-defined string and current item procedures are set to do-nothing routines.

If the constructor fails, check the InitStatus variable for a detailed error code.

Examples

```
if not M.InitCustom(2, 2, ScreenWidth-1, 2,          {Menu coordinates}
                    DefaultColorSet,                 {ColorSet}
                    DefWindowOptions or wBordered,   {Window options}
                    Horizontal)                      {Menu orientation}
then begin
  WriteLn('Failed to Init Menu,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates a framed, horizontal menu using default color options.

```
if not M.InitCustom(1, 1, ScreenWidth, 1,          {Menu coordinates}
                    DefaultColorSet,               {ColorSet}
                    DefWindowOptions or wBordered, {Window options}
                    Horizontal)                    {Menu orientation}
then begin
  WriteLn('Failed to Init Menu,  Status = ', InitStatus);
  Halt;
end;
```

Incorrect! A window frame was requested but the window dimensions do leave enough room for a frame.

See Also

CommandWindow.InitCustom Init

Declaration
```
function IsCurrent : Boolean; virtual;
```

Purpose
Return True if menu system is the current window.

Description
Works like RawWindow.IsCurrent except that it returns True if the IsCurrent method of any part of the menu returns True.

Example
```
if M.IsCurrent then
  M.Erase;
```
Erase a menu system if one of its submenus (or the top level menu) is the current window.

See Also
RawWindow.IsCurrent

Declaration
```
procedure ItemsDone;
```

Purpose
Terminate the current item list. Set first item as default.

Description
You must call this method (or ItemsDoneCustom) after adding all items to a particular submenu. OPMENU then pops an internal stack so that further items will be added to the previously active submenu.

See "Building a Menu System" for more information and a complete example program.

Example
```
M.AddFramedSubMenu(4, 4, 20, 7, Vertical, DefWindowFrame);
  M.AddItem('New',  1, 1, 2);
  M.AddItem('Save', 2, 1, 3);
  M.AddItem('Dos',  3, 1, 20);
  M.ItemsDone;
```
Creates a submenu, adds three items, and closes the submenu with the call to ItemsDone. The first item on the menu (New), will be highlighted when this submenu is selected. Because ItemsDone doesn't optimize the size of the submenu, the window will have a blank row at the bottom (it has four rows and only three items).

See Also
ItemsDoneCustom

ItemsDoneCustom

Declaration

```
procedure ItemsDoneCustom(OptimizeSize : Boolean; DefaultKey : LongInt);
```

Purpose

Terminate the current item list and customize size and default key.

Description

You must call this method (or ItemsDone) after adding all items to a particular submenu. OPMENU then pops an internal stack so that further items will be added to the previously active submenu. If the OptimizeSize parameter is True (and the UseAdjustableWindows conditional compilation define is enabled in OPDEFINE.INC), ItemsDoneCustom resizes the submenu to surround the items exactly. It bases the size on the orientation of the submenu, the length of each item name, and the length of the selection markers and item padding at the time of the call.

DefaultKey specifies the item to be the default selection of the submenu. If the specified key isn't found, the first unprotected item in the submenu becomes the default. If the specified item is protected, the closest previous item becomes the default. The constant mnNotAKey (which equals -1) may be passed in this position to select the first item as the default.

See "Building a Menu System" for more information.

Example

```
M.AddFramedSubMenu(4, 4, 20, 10, Vertical, DefWindowFrame);
  M.AddItem('New',       1, 1, 2);
  M.AddItem('Save    ', 2, 1, 3);
  M.AddItem('Dos',       3, 1, 4);
  M.ItemsDoneCustom(True, 4);
```

Creates a submenu, adds three items, and terminates the submenu with the call to ItemsDoneCustom. The submenu window will be resized to exactly fit the items we've added. Note that the sizing respects the blank spaces at the end of the 'Save' string. By padding one item in the submenu you can reserve space for status information to be patched in by a custom string procedure. 'Dos' will be highlighted when this submenu is first activated.

See Also

DefaultItem
ItemsDone
SetLeftPad
SetSelectMarker

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a Menu from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Menu.Store. Load first calls CommandWindow.Load to read the underlying CommandWindow. Then it reads data specific to the Menu. This includes a potentially recursive sequence of calls to read the menu items, their submenus, and their items.

The stream registration procedure for a Menu is MenuStream. There are also two procedure pointers (the custom string and current item routines) that may need to be registered independently. The example program below demonstrates how. See Store for more details.

Example

```
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpMenu;
var
  M : Menu;
  S : BufIdStream;

  {$F+}
  procedure CurrentItemProc(CurrentItemPtr : MenuItemNodePtr; MPtr : MenuPtr);
  begin
    FastWrite(Pad(CurrentItemPtr^.HelpString, ScreenWidth),
              ScreenHeight, 1, TextAttr);
  end;
  {$F-}

begin
  {Load the menu system from a stream}
  S.Init('MENU.STM', SOpen, 1024);
  S.RegisterHier(MenuStream);
  S.RegisterPointer(1000, @CurrentItemProc);
  S.Get(M);
  S.Done;
  M.Process;
  M.Done;
end.
```

Instantiates M from the stream created by the Store example. Note that the address of the current item procedure is registered with the same user code as that written by Store. S.Get calls Menu.Load indirectly.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

CommandWindow.Load Store

MenuChoice / mnOptionsAreOn

Declaration

```
function MenuChoice : LongInt;
```

Purpose

Return the last menu choice.

Description

Call MenuChoice only after calling Process. MenuChoice returns the key of the item selected by the user.

Example

```
repeat
  M.Process;
  case M.GetLastCommand of
    ccSelect :
      case M.MenuChoice of
        1 : {Perform action for menu item 1};
        2 : {Perform action for menu item 2};
        {...}
      end;
    ccError :
      Finished := True;
    ccQuit :
      {User pressed <Esc> at top level. Perhaps just ignore this command.}
      ;
    {...}
  end;
until Finished;
```

Shows how MenuChoice is used to detect what menu item was selected. GetLastCommand should always be checked first to assure that the user selected an item and didn't escape out of the menu system or press some other exit command.

See Also

Process

Declaration

```
function mnOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This function tests the menu options, which are described in the Constant declarations near the beginning of this section.

Example

```
if M.mnOptionsAreOn(mnAlphaMatch) then
  {take some action};
```

Takes some action if the mnAlphaMatch option is on.

See Also

mnOptionsOff mnOptionsOn

Declaration
```
procedure mnOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadMenuOptions (see the Constants section, above).

Example
```
M.mnOptionsOff(mnAlphaMatch+mnArrowSelect);
```
Turns off options to select a menu item matching an alpha character and to treat arrow keys in an extended fashion.

Declaration
```
procedure mnOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadMenuOptions (see the Constants section, above).

Example
```
M.mnOptionsOn(mnUseItemForTopic+mnSelectOnClick);
```
Enables options to use the current item key for the context-sensitive help hook and to select items based on a mouse click.

Declaration
```
{$IFDEF UseAdjustableWindows}
procedure MoveWindow(DX, DY : Integer); virtual;
```

Purpose
Move a menu.

Description
This method overrides StackWindow.MoveWindow so that it simultaneously shifts all windows (active and inactive) that comprise a menu system. The menu system must either be inactive or contain the current window for MoveWindow to work; otherwise it generates the error epNonFatal+ecWinNotCurrent. If the menu system is on-screen, MoveWindow allocates a temporary virtual screen to perform the updates. If there is insufficient memory for the virtual screen, it generates the error epNonFatal+ecOutOfMemory and exits before modifying the menu system.

MoveWindow visits all windows of the menu system and attempts to adjust their positions by the specified number of columns (DX) and rows (DY). If doing so would move the window outside of its specified coordinate limits, MoveWindow moves the window to the appropriate limit and does not generate an error.

Example
```
M.MoveWindow(+1, +1);
```
Moves the entire menu system one column right and one row down. If this would push any submenu off the edge of the screen, MoveWindow moves that submenu less.

See Also
StackWindow.MoveWindow

Process / ProtectItem

Declaration

```
procedure Process; virtual;
```

Purpose

Process menu commands leading to a choice.

Description

The Menu Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Menu Commands" at the start of this section.

It turns on the options mnMainSelect and mnMainHighlight before updating the menu and starting the command processing loop.

Before exiting, Process stores the most recently selected item in an instance variable where it can be accessed by the MenuChoice method. If the mnPopOnSelect option is enabled and the user exited with the ccSelect command, Process erases the active submenu (but not the top level menu) before exiting.

Process may call any of several user-defined routines. The custom string procedure and the current item procedure are specific to the Menu. The error handling procedure is inherited from the CommandWindow. And the context-sensitive help and keyboard background processing procedures are inherited from the CommandProcessor.

Example

For a complete example, see the introduction to this section.

See Also

CommandWindow.Process MenuChoice

Declaration

```
procedure ProtectItem(Key : LongInt);
```

Purpose

Protect specified item. No screen update.

Description

If no item matching the specified key is found, ProtectItem exits without producing an error. If the specified item is a selected item, ProtectItem tries to move the selection to the nearest previous unprotected item. Note that if the specified item is selected and has an attached submenu that is also visible, ProtectItem does not take the steps needed to deselect the submenu. It is a programming error to protect an item in this circumstance.

See the discussion of "Protected Items" in the introduction for more information.

Example

```
M.ProtectItem(9);
M.ProtectItem(10);
M.Process;
```

Protects menu items with keys 9 and 10 before processing the menu.

See Also

UnprotectItem

Declaration

```
procedure Redraw;
```

Purpose

Erase and redraw menu system, presumably after changing an item.

Description

OPMENU activates a temporary virtual screen to avoid flickering during the update. Because the virtual screen is used, it is acceptable for other windows to be drawn on top of the menu system at the time Redraw is called. However, these windows must meet the requirements specified in StackWindow.ActivateWrite.

The most common times to call Redraw are after protecting or unprotecting items, or after changing status variables that affect a menu's custom string procedure.

Example

```
M.Process;
case M.GetLastCommand of
  ccSelect :
    case M.MenuChoice of
      {...process choices...}
      99 :
        begin
          if IsAdvanced then begin
            M.ProtectItem(10);
            M.ProtectItem(11);
          end else begin
            M.UnprotectItem(10);
            M.UnprotectItem(11);
          end;
          M.Redraw;
        end;
    end;
  {...}
end;
```

Item 99 of the menu system is presumed to be one that toggles a menu system between "advanced" and "beginner's" modes. After changing the protection, it redraws the menu system. Process itself would update only the current submenu.

See Also

Erase

SelectItem

Declaration

```
procedure SelectItem(Key : LongInt);
```

Purpose

Select specified item and update screen.

Description

SelectItem first searches the menu system to find the specified Key. If the key is not found or if the item is protected, SelectItem exits with no action. Otherwise, if the menu system is not active, SelectItem displays all submenus leading to the specified item and highlights that item. If the menu system is already active, SelectItem erases any submenus that don't lead to the item and draws any new submenus required to reach it. This sequence does not occur on a virtual screen, so a brief flicker may be visible if a significant change is required to reach the specified item.

The next call to Process will proceed from the selected item.

Examples

```
InitMenu(M);
M.SelectItem(10);
M.Process;
```

Sets item 10 as the initial selection and calls Process to allow the user to confirm or change the choice.

```
repeat
  M.Process;
  case M.GetLastCommand of
    ccSelect :
      {...process choices...} ;
    ccUser0 :
      M.SelectItem(10);
    {...}
  end;
until Finished;
```

ccUser0 can be assigned to a hot-key, which selects menu item 10 with a single keystroke.

See Also

DefaultItem SelectSubMenu

Declaration

```
function SelectItemByPos(X, Y : Byte) : Boolean;
```

Purpose

Select visible item (if any) at absolute position and update screen.

Description

This method is useful in situations where the mouse has been enabled and a menu system remains on-screen while other windows are active. (DESKPOP is an example; it provided the impetus for writing this method.) If the specified coordinates are within the text of an unprotected menu item, SelectItemByPos will select that menu item and return True. Otherwise it does nothing but return False.

SelectItemByPos assumes that the menu is not partially obscured by other windows. If it is, those windows will be overwritten when SelectItemByPos updates the menu system. If the possibility exists for other windows to obscure the menu system, erase them before calling SelectItemByPos. Note that DESKPOP avoids this situation by reserving the top row of the screen for a menu bar. SelectItemByPos only applies to this top level menu, and the remaining window types never overwrite the top row.

Example

```
var
  MX, MY : Byte;
  Status : ButtonStatus;
  WPtr : WindowPtr;
  M : Menu;
begin
  ...
  MouseWhereXYabs(MX, MY, Status);
  WPtr := wStack.EvaluateStackPos(MX, MY);
  if (WPtr = nil) or (WPtr = @M) then
    {Mouse is not within any window but perhaps the top level menu}
    if M.SelectItemByPos(MX, MY) then begin
      {The menu system is activated, get a choice}
      M.Process;
      ...
    end;
```

The call to MouseWhereXYabs occurs after a menu system has been displayed, one or more other windows have been drawn, and a mouse click outside of the current window has been detected. EvaluateStackPos scans the default window stack to determine which window, if any, the mouse cursor is in. If the returned pointer is nil, the mouse cursor is within no visible StackWindow (which does not preclude the possibility of it being within a submenu, which isn't managed by the default window stack); if the returned pointer is @M, then the cursor is within the top level menu. SelectItemByPos attempts to select a menu item within any active portion of the menu system. If it succeeds it updates the screen to show that item selected. At that point the logical step is to call Menu.Process so that the user can make a menu selection.

See Also

SelectItem

WindowStack.EvaluateStackPos

SelectSubMenu / SetAllXxxAttr

Declaration

```
procedure SelectSubMenu(Key : LongInt);
```

Purpose

Select current item of submenu containing Key and update screen.

Description

Works just like SelectItem, except that the current item of the most deeply nested matching submenu is not changed. Key serves only to identify a specific submenu; its value can match any item in the submenu. By convention, we specify the first item in the submenu to pass to SelectSubMenu.

Example

```
M.SelectSubMenu(11);
```

Selects the default item of the submenu that contains item 11.

See Also

DefaultItem SelectItem

Declaration

```
procedure SetAllXxxAttr(Color, Mono : Byte);
```

Purpose

Set specified attributes for all windows in the menu system.

Description

This entry describes a group of related methods that change a specified video attribute for all submenus in a menu system. You should call these methods only after the entire menu system has been defined. The actual methods are:

```
procedure SetAllFrameAttr(Color, Mono : Byte);
procedure SetAllHeaderAttr(Color, Mono : Byte);
procedure SetAllHighAttr(Color, Mono : Byte);
procedure SetAllNormAttr(Color, Mono : Byte);
procedure SetAllProtectAttr(Color, Mono : Byte);
procedure SetAllSelectAttr(Color, Mono : Byte);
{$IFDEF UseShadows}
procedure SetAllShadowAttr(Color, Mono : Byte);
{$ENDIF}
```

Color is the video attribute to use in a color video mode; Mono is the attribute for a monochrome mode.

See the discussion of "Menu Customization" near the beginning of this section for more information.

Example

```
M.SetAllHighAttr($0A, $70);
```

Sets the highlight attribute–applied to characters used for alphanumeric searching–for all menu windows to light green on black for color monitors, reverse video for monochrome monitors.

See Also

SetHelpAttr SetXxxAttr

Declaration

```
procedure SetAllSelectMarker(Left, Right : String);
```

Purpose

Define strings to mark selected items in all submenus.

Description

SetAllSelectMarker visits all item lists in the menu system and calls SetSelectMarker for each one. You should call this method only after the entire menu system has been defined.

Example

```
M.SetAllSelectMarker('< ', ' >');
```

Sets the selected markers for the main menu and all submenus.

See Also

SetSelectMarker

Declaration

```
procedure SetCurrentItemProc(IP : mnCurrentItemProc);
```

Purpose

Set procedure to be called whenever the current item is drawn.

Description

Whenever OPMENU moves to a new item, selects a submenu, or erases a submenu, it calls a virtual method (CurrentItem). The virtual method just calls another procedure using a procedure variable. SetCurrentItemProc specifies that a particular user-written procedure will gain control at this time.

The procedure passed as a parameter to SetCurrentItemProc must be compiled using the FAR model (with the {$F+} compiler directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
procedure CurrentItemProc(CurrentItem : MenuItemNodePtr; MPtr : MenuPtr);
```

See the discussion of "The Current Item Procedure" in the introduction to this section for more details and a complete example. Also note that when you enable MAKEMENU's help row capability, the source code it generates includes a current item procedure that is used to update the help row for the current item.

See Also

CurrentItem SetCustomStringProc

SetCustomStringProc / SetHelpAttr

Declaration

```
procedure SetCustomStringProc(CP : mnCustomProc);
```

Purpose

Set item string customization procedure.

Description

Before it displays each item, OPMENU transfers control to a user-written routine called the custom string procedure. This procedure may modify the displayed string to include status information specific to the item. The default string procedure does nothing. SetCustomStringProc specifies that a particular user-written procedure will gain control at this time.

The procedure passed as a parameter to SetCustomStringProc must be compiled using the FAR model (with the {$F+} compiler directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
procedure CustomStringProc(var Name : String; Key : LongInt;
                           Selected, Highlighted : Boolean;
                           WPtr : RawWindowPtr);
```

See the discussion of "The Custom String Procedure" in the introduction to this section for more details and a complete example. Note that MAKEMENU never generates a custom string procedure for you, since the entire implementation must be intimately tied to a particular application.

See Also

SetCurrentItemProc

Declaration

```
procedure SetHelpAttr(Color, Mono : Byte);
```

Purpose

Set the colors for the help line.

Description

This routine simply stores the specified attributes in data fields of the menu object. OPMENU never actually uses these colors; it stores them simply for the convenience of user-written current item routines.

See Also

GetHelpAttr

Declaration

```
procedure SetXxxAttr(Color, Mono : Byte);
```

Purpose

Set specified attributes for the current item list.

Description

This entry describes a group of related methods that change a specified video attribute for the current submenu's item list or frame. You should call these methods just after the submenu has been added to the menu system. The actual methods are:

```
procedure SetHighAttr(Color, Mono : Byte);
procedure SetNormAttr(Color, Mono : Byte);
procedure SetProtectAttr(Color, Mono : Byte);
procedure SetSelectAttr(Color, Mono : Byte);
procedure SetFrameAttr(Color, Mono : Byte);
procedure SetHeaderAttr(Color, Mono : Byte);
{$IFDEF UseShadows}
procedure SetShadowAttr(Color, Mono : Byte);
{$ENDIF}
```

Color is the video attribute to use in a color video mode; Mono is the attribute for a monochrome mode.

See the discussion of "Menu Customization" near the beginning of this section for more information.

Example

```
M.AddFramedSubMenu(9, 4, 15, 5, Vertical, Frame2);
M.SetNormAttr($02, $07);
```

Sets the normal text attribute to green on black for color monitors, normal video for monochrome monitors. These colors only apply to the submenu just added with AddFramedSubMenu.

See Also

SetAllXxxAttr

SetLeftPad / SetSelectMarker

Declaration

```
procedure SetLeftPad(LeftPad : Byte);
```

Purpose

Set the number of characters to left pad each item when displayed.

Description

This method applies only to the submenu most recently added during initialization of a menu system.

Note that vertical submenus are left-padded with one character by default; horizontal submenus are not padded by default. In addition, LeftPad characters are added to the *right* side of each item name in a vertical submenu after the string is returned by the string customization routine. This practice results in more pleasingly symmetrical vertical menu displays.

See the discussion of "Menu Customization" near the beginning of this section for more information.

Examples

```
M.SetLeftPad(2);
```

Inserts two spaces before each menu item.

```
M.AddFramedSubMenu(9, 4, 15, 5, Vertical, Frame2);
M.SetLeftPad(0);
```

Removes all padding for items on the submenu just added.

Declaration

```
procedure SetSelectMarker(Left, Right : String);
```

Purpose

Define strings to mark selected items.

Description

SetSelectMarker stores the specified strings in data fields of the active submenu's item list. Note that these data fields are capable of holding a string at most 5 characters long.

When OPMENU draws a selected item it concatenates these strings to the start and end of the item's name.

Example

```
M.SetSelectMarker('[', ']');
```

Surround the selected item with brackets.

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a list in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the Menu to the stream S. It starts by calling CommandWindow.Store, then continues to write the data specific to a Menu. The menu items may require a recursive sequence of writes to store attached submenus and their items.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

The stream registration procedure for a Menu is MenuStream. If custom string or current item procedures are being used, their addresses must also be registered independently. The example program below demonstrates how.

If the Menu uses any but the default CommandProcessor (MenuCommands), the address of that object must be registered. See CommandWindow.Store for an example.

Example

```
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpMenu;
var
  M : Menu;
  S : BufIdStream;

  {$F+}
  procedure CurrentItemProc(CurrentItemPtr : MenuItemNodePtr; MPtr : MenuPtr);
  begin
    FastWrite(Pad(CurrentItemPtr^.HelpString, ScreenWidth),
              ScreenHeight, 1, TextAttr);
  end;
  {$F-}

  procedure InitMenu(var M : Menu);
  begin
    with M do begin
      InitCustom(2, 2, ScreenWidth-1, 2, DefaultColorSet,
                 DefWindowOptions or wBordered, Horizontal);
      AddItemHelp('File', 4, 1, 1, 'New/Save files');
      AddFramedSubMenu(4, 4, 10, 6, Vertical, DefWindowFrame);
        AddItemHelp('New',  1, 1, 2, 'Create a new file');
        AddItemHelp('Save', 2, 1, 3, 'Save the current file');
        AddItemHelp('Dos',  3, 1, 20, 'Shell to DOS');
        ItemsDone;
      AddItemHelp('Edit',    20, 1, 4, 'Edit the current file');
      AddItemHelp('Options', 35, 1, 5, 'Set program options');
      ItemsDone;
      SetCurrentItemProc(CurrentItemProc);
    end;
  end;

begin
  InitMenu(M);
  {Store the menu system to a stream}
  S.Init('MENU.STM', SCreate , 1024);
```

```
    S.RegisterHier(MenuStream);
    S.RegisterPointer(1000, @CurrentItemProc);
    S.Put(M);
    S.Done;
    M.Done;
  end.
```

Builds a small menu system, M, and stores it to a stream. See the Load example for a program to read this stream back in. Note that the address of the current item procedure is registered with a unique user code.

See BufIdStream (in OPROOT) for proper error checking using streams.

See Also

CommandWindow.Store Load

Declaration

```
procedure UnprotectItem(Key : LongInt);
```

Purpose

Unprotect specified item. No screen update.

Description

If no item matching the specified key is found, UnprotectItem exits without producing an error. Otherwise it resets the item's protected flag and exits. Note that all items are unprotected by default.

See the discussion of "Protected Items" in the introduction for more information.

Example

```
M.UnprotectItem(9);
M.UnprotectItem(10);
M.Process;
```

Unprotects menu items with keys 9 and 10, which resumably were protected earlier.

See Also

ProtectItem

Declaration

```
{$IFDEF UseAdjustableWindows}
procedure UpdateScreenSize;
```

Purpose

Update window size, position, and limits for current screen size.

Description

This method works just like RawWindow.UpdateScreenSize, except that it is applied to all windows that comprise the menu system.

See Also

RawWindow.UpdateScreenSize

VisitAllItems

Declaration

```
procedure VisitAllItems(IProc : mnItemProc; var D);
```

Purpose

Call the specified procedure for all items in the menu system.

Description

ItemProc must be compiled under the FAR model (with the {$F+} compiler directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
procedure ItemProc(IPtr : MenuItemNodePtr; var D; MPtr : MenuPtr);
```

The untyped var parameter D is provided for the convenience of the user-written procedure. Also passed to the ItemProc are the address of the visited item and the address of the menu system. VisitAllItems visits the items in depth-first, first-to-last order, starting with the top level menu's first item. It visits protected items.

The ItemProc may perform any action that does not corrupt the structure of the menu tree. MAKEMENU calls VisitAllItems to implement the "Rekey" command, which sequentially renumbers all items in depth-first order.

Example

```
var
  M : Menu;
  B : Byte;

  {$F+}
  procedure ChangeAllSelectPos(IPtr : MenuItemNodePtr; var D; MPtr : MenuPtr);
  begin
    IPtr^.ChangeSelectPos(Byte(D));
  end;
  {$F-}
  ...
begin
  InitMenu(M);
  B := 0;
  M.VisitAllItems(ChangeAllSelectPos, B);
end.
```

Changes the select character of all menu items by visiting all items and calling the ChangeAllSelectPos procedure. This example changes all select characters to zero (no select character).

See Also

VisitAllWindows

Declaration

```
procedure VisitAllWindows(WProc : mnWindowProc; var D);
```

Purpose

Call the specified procedure for all submenus in the menu system.

Description

WProc must be compiled under the FAR model (with the {$F+} compiler directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
procedure WindowProc(WPtr : RawWindowPtr; var D; MPtr : MenuPtr);
```

The untyped var parameter D is provided for the convenience of the user-written procedure. Also passed to the WindowProc are the address of the visited window and the address of the menu system. VisitAllWindows visits the windows in depth-first, first-to-last order, starting with the Menu object itself.

The WindowProc may perform any action that does not corrupt the structure of the menu tree. OPMENU itself calls VisitAllWindows to implement EnableAllExplosions, EnableAllAlternateOpen, EnableAllNormalOpen, AddShadows, IsCurrent, and other methods.

Example

```
var
  M : Menu;
  B : Boolean;

  {$F+}
  procedure SetSoundOption(WPtr : RawWindowPtr; var D; MPtr : MenuPtr);
  begin
    if Boolean(D) then
      WPtr^.wOptionsOn(wSoundEffects)
    else
      WPtr^.wOptionsOff(wSoundEffects);
  end;
  {$F-}
  ...
begin
  InitMenu(M);
  B := True;
  M.VisitAllWindows(SetSoundOption, B);
end.
```

Changes the sound option of all menu windows by visiting each window with the procedure SetSoundOption. In this case, sound is turned on in all windows.

Note that the WindowProc can distinquish between the top level menu's window and the submenu windows by using the following statement:

```
if WPtr = RawWindowPtr(MPtr) then
  {This is the top level window}
else
  {This is a submenu's window};
```

See Also

VisitAllItems

Index

0

8087 reals 11-17
8x8 font 3-22, 35
<CtrlBreak> 4-152
{$N-} 2-9
{$V-} 6-112, 6-176

A

Action routines (for selectors) 6-85
Active coordinates (for windows) 4-70
Active window 4-56
Alternate draw 4-75
Array 8-154
Array, page structure 8-166
Array, priorities 8-157
Array, optimized 8-154
Array, performance 8-156
Array, range checking 8-158
ArrayField 6-29
ASCIIZ strings 9-35
Auto-advance 6-22
Auto-execution of commands 3-91

B

Background task hook 3-89
Backward links 6-74
BCD 14-2
BcdField 6-29
BEST 6-249
Binary Coded Decimal 14-2
Bit flags 14-28
Bit sets 8-50
Blinking attribute 3-6, 3-36
Block copy hook (for help) 4-411
Block function (for printer) 7-80
Block-oriented commands 5-1
BooleanField 6-30
Border 4-8, 4-21, 4-50
Box field 7-70
Boyer-Moore algorithm 9-4
BROWSE 5-80
BrowseStatusProc 5-63
Browser commands 5-57
Browsing 5-1
Burns, Philip R. 14-3
Buttons (see Mouse, buttons)
ByteField 6-30

C

C. Itoh 8510 printer 7-59
Calculated fields 6-124
Calculator 1-30, 14-3
Calendar 1-35
CALIBRAT.LZH 7-6
Calibration, printer 7-6
CGA 3-7
CharEditor 6-52, 6-119, 6-181
CharField 6-31
Character pitch 7-27
Character point 7-27
Characters per inch 7-27
CheckBreak 3-6
CheckEOF 3-6
CheckSnow 3-6
ChoiceField 6-31
Circular linked list 8-29
Cloning 14-16
Cluster size 4-292
Code numbers for objects 8-96
Color names 3-49
ColorMap 3-8
ColorSet 2-8, 3-45
ColorSet and streams 4-160
Commands, codes 3-83
Commands, general 4-140
Commands, keyboard handling 2-8
CompField 6-32
Compiling large programs 15-24
CompuServe 1-45, title-page
Conditional defines 2-9, 4-22, 4-57
Configuration data 2-8, 14-16
Constructors 2-5
Context sensitive help 2-8, 3-90, 4-138, 4-141, 4-312, 4-361, 4-399
Conventions (for this manual) 2-4
Conversion routine (for data entry) 6-15, 6-141
Conversion routine (for printer) 7-77
Copy file 10-6
Cross-reference (for help) 4-367
CRT 3-3
Current field (for data entry) 6-59
Current item procedure (for PickList) 4-312, 4-330
Custom string procedure (for PickList) 4-309
Customizing programs 14-16

D

Data entry 1-2, 6-1
Data entry, commands 6-136
Data entry, fields 6-15
Data entry, nested screens 6-127
Data entry, overview 6-140
Data entry, screens 6-110
Data objects 1-2, 8-1
DataProducts DP 7-59
Date variables 6-206
Date, valid ranges 6-207

DateField 6-32
DateStField 6-33
Decipoint 7-28
Derived types 2-4
DESKPOP 1-24
DesqView 3-1, 3-8
Destructors 2-5
Diablo daisywheel printer 7-27, 7-58
DirectVideo 3-6
Directory 10-18
Directory, changing 4-253
Directory, commands 4-243
Directory, entry (for libraries) 8-127
Directory, format 4-278
Directory, list 4-241
Directory, sorting 4-246
Directory, tree format 4-282, 4-293
Directory, window 4-281
Disk transfer address (DTA) 10-1
Disk class 10-13
Disk sectors 10-22, 10-31
Divider bars 4-39
DPINST 1-40
DOS critical error handler 11-16
DOS environment 10-1
DOS services 11-15
DOS services for printer output 7-4
DOS shell 10-27
DOS unit 10-1
DosBusyFlag 11-27
DosCriticalFlag 11-27
DoubleField 6-33
Doubly linked list 8-29
Draw, menu 4-306
Draw, window 4-72

E

EDITOR 5-79
Editor 1-33, 5-19
EGA 3-1, 3-7, 3-22, 3-35
EMS 8-154, 12-1, 12-3
Enhanced keyboards 3-5, 10-2, 10-42
ENTRY 6-246
Entry screens 6-1
Entry screens and streams 6-133
Entry screens, wrap mode 6-62, 6-80
EntryField 6-20
Epson LQ printers 7-58
Epson MX printers 7-58
Epson dot matrix printers 7-27, 7-58
Error categories 8-150
Error codes 4-141, 8-150, 15-10
Error handling 2-7, 4-140, 7-79, 8-150, 8-158
Error handling, default 4-156
Error message strings 8-150
Error types 15-10
Exchanging values 14-32
ExecDos 10-1
Exit command 6-141, 4-152, 4-157
Expanded memory (see EMS)
Exploding windows 4-77, 4-333
Extended memory 12-1
ExtendedField 6-33
Extension (file) 9-11, 9-13, 9-18
External Interfaces (for TSRs) 11-19, 11-41

F

Facit 4512 printer 7-59
Far calls 14-34
FastDispose 2-9
Field, adding 6-114
Field, attributes 6-58
Field, conversion routines 6-40, 7-76
Field, definitions 6-4
Field, delimiters 6-123
Field, derived types 6-29
Field, drawing routines 6-48
Field, editors 6-4, 6-15, 6-52, 6-141
Field, hidden 6-56, 6-58
Field, ID 6-55, 7-96
Field, invisible 6-57
Field, links 6-60
Field, lists 6-55
Field, movement commands 6-60, 142
Field, options (for data entry) 6-21, 6-122, 7-86
Field, protected 6-56, 6-58
Field, selecting 6-1, 6-3, 6-55, 6-58
Field, user-defined 6-130
Field, validation routines 6-43, 6-44
File attribute 4-242
File browser 5-1, 5-56
Filename 9-18
Filename filtering 4-244
Filename formatting 4-246
Flags 14-44
Float 9-2
Font 7-28, 7-32
Font, stack 7-32
Font, changing 7-27
Font8x8 3-6
Footers 7-78
Formatted strings 9-14, 9-21, 9-49
FORMTEST 7-113
FourByteDates 2-9
Frame 4-8, 4-20 (see Window)
Frame-relative writing 4-81, 4-114
Free pool, window numbers 4-129

G

Generating documentation 6-244

Generating menus 4-420
Generating source code 6-244
Global validation routine 6-126, 6-156
GRAPH3 3-3
GUIDE 1-17

H
Hashing 8-68
Header, more (for PickLists) 4-208, 4-384
Header, printer 7-78
Header, spanning 4-39
Header, window 4-21
Heap usage 2-7
Help 1-17, 4-361
Help, bound in 4-372, 4-394
Help, context-sensitive 2-8, 3-90, 4-138, 4-141, 4-312, 4-361, 4-399
Help, cut/paste 1-21, 4-411
Help, file 4-375
Help, file control characters 4-367
Help, file directives 4-364
Help, index 4-138, 4-365
Help, paged 4-406
Help, previous topic stack 4-377, 4-405
Help, scrolling text 4-409
Help, topics 4-298, 4-365
Hexadecimal string 9-16
HP DeskJet 7-58
HP LaserJet 7-27
Hot spot 4-25
Hot spot, for help 4-377
HotCode 4-91
Hotkeys 11-13
HugePtr 12-12

I
ICD Files 2-8
InitAllUnits 2-9
InitStatus 8-151
Inline macros 14-1, 28
Inline procedures 13-1
INSTALL.EXE 1-10
Installation 1-7, 1-10
IntegerField 6-34
International currencies/conventions 6-11
Interrupt service routines 11-1
Interrupt vector 11-1, 11-4
Interrupts and TSRs 1-3
Item key (for menus) 4-298

J
Julian date 6-208
Jump marker 14-28

K
Key assignment 3-87, 3-112
Key codes 10-42
Keyboard macros 1-3, 13-1
Keyboard status flags 3-27
Keystrokes, commands 3-82

L
LHARC 1-5, 14-28
Landscape mode 7-28
Large array (see Array)
LaserJet printers 7-33, 7-58
LastMode 3-6
Leading 7-28
Letter sets 9-8
Library 8-127
Library directory 8-127
Line editor 6-1, 6-176
Linked list 8-29
Literals (see Picture masks)
LoadableColorSet 4-2
LongIntField 6-34
Lotus style menus 4-297

M
Macro, chaining 13-12, 13-21
Macro, editor 13-15
Macro, engine 13-1, 13-8
Macro, file 13-11, 23
Macro, nested 13-1
Macro, playback 13-18
Macro, recording 13-10
Macro, scrap 13-18
Make file (installation) 1-7
MAKEHELP 4-369
MAKEMENU 4-420
MAKESCRN 6-228
Mapping, video attributes 3-28
Mask (see Picture mask)
Mask, characters 6-5
Mask, date 6-6
Mask, directory header 4-251
MCGA 3-7
Memo 5-1, 5-79
Memo field 5-2, 5-6
Memo window 5-22
Memory allocation 1-3, 12-1
Memory resident programs (see TSR)
Menu, building 4-298
Menu, commands 4-304
Menu, customization 4-301
Menu, Lotus style 4-297
Menu, protected items 4-203, 4-308
Menu, systems 4-297
MergePicture 6-14

Methods, static 2-4
Methods, virtual 2-4
Mickeys 3-60
Micronics printer 7-58
MODE 7-5
Modules (installable) 11-19
Mono 3-6
MonoHerc 3-7
Mouse, buttons 3-54, 3-68
Mouse, coordinates 3-55
Mouse, cursor 3-53
Mouse, drivers 3-59
Mouse, droppings 3-56
Mouse, event Handlers 3-57
Mouse, support 2-8, 3-1, 3-88
Mouse, windows 3-55
MultiLineField 6-35, 6-125, 6-128
Multiple choice lists 4-196
Multitasking system 3-8

N

Naming conventions (for this manual) 2-6
NEC 3510/3530 printers 7-58
NEC 7710/7730/8810/8830 printers 7-59
NestedField 6-35
NullValidation 6-44
NumberEditor 6-52, 120, 181
Numeric coprocessor 11-3
Numeric field 6-21

O

Object Professional Library 8-129
Object, code numbers 8-96
Object, hierarchy 15-7
Object, version number 8-96
Object-oriented I/O 8-90
Object-oriented data structures 8-2
Octal string 9-23
OmniView 3-1, 3-8
OPL 8-129
OS 14-28
Overlays 11-17, 15-14

P

Packed image 4-165
Packed window 4-436
Packed window, bound in 4-168
PackedWindow 4-164
Page frame 12-7 (see EMS)
Panasonic KX-P1093 printer 7-58
Pasting help text 1-21
Path list 4-241
Pathname 9-7, 9-16, 9-19
Periscope 11-16
PGC 3-7
PickHorizontal 4-194
PickList 4-186
PickList field 6-128
PickList, orientations 4-194
PickList, user move hook 4-203
PickListFields 2-10
PickSnaking 4-194
PickVertical 4-194
Picture mask (for printing) 7-2
Picture mask 6-4, 141, 241
Picture mask, semi-literal characters 6-5
Picture mask, user defined characters 6-6
Pitch, character 7-27
PlaybackDelay 13-3
Point 7-28
Point, character 7-27
Pointer arithmetic 14-1
PointerStack 8-6
POPHELP 1-17, 11-35, 13-23
Popup, dispatcher 11-13
Popup, handles 11-14
Popup, routines 11-13, 14
Portrait mode 7-28
Post-edit routines 6-41, 6-124, 6-141, 6-150
Post-select routines 6-86
Pre-edit routine 6-113, 6-123, 141
Pre-select Routines 6-86
PRINT.COM 10-1
Printing 1-2, 7-1
Printing, calibration 7-6
Printing, custom hook
Printing, error handling 7-1, 7-34
Printing, fields 7-2
Printing, forms 7-66
Printing, hardware registration 7-57
Printing, lines and boxes 7-27
Printing, modes 7-2
Printing, overview 7-84
Printing, page function 7-80
Printing, printer libraries 7-59
Printing, streams 7-35, 7-81
Printing, user data field 7-98
Printing, user hooks 7-79
Procedure pointers 2-4
Program customization 14-16
Proprinter II printer 7-58
Protected field 6-56
PSCREEN 4-436
Pseudo-scan code 3-71
PTIME 11-64
Pulldown menus 4-297
Purchase Agreement 1-45

Q

Queues 8-19
Quick Reference Chart 1-38

Quicksort 14-1

R

Range checking (large arrays) 8-158
READ.ME 1-5
Reading/writing text files 5-19
RealField 6-36
Regular field 6-21
Reject functions (for DirList) 4-245
Replay 13-2
Reports 7-68
Required fields 6-126
Run-length compression 4-165
Runtime error recovery 14-25
Runtime errors 11-16

S

SaveInt1B 3-6
Saving, windows 4-112
Scan code 3-31
Scrap macro 13-18
Screen rectangles 4-9
Screen update routines 6-61
Screen/keyboard Handling 1-1
Scroll bars 4-21, 4-25, 4-113
Scrolling, windows 4-112
Search attribute (for files) 4-274
Search mask (for files) 4-242, 4-274
Search string header (for files) 4-209
Search-and-replace 5-1
Secondary command tables 3-90
Secondary field options 6-24
SelectField 6-17
Selecting fields 6-3, 6-58
Selecting fields, Action Routines 6-85
Semi-literal characters 6-5
Separator bar 4-328
Shadows 4-21, 24
ShortIntField 6-36
SideKick 11-21
Simple field 6-21
Simple line editor 6-197
SimpleCharEditor 6-52, 6-120, 6-179
SimpleStringEditor 6-52, 6-118, 6-177
SingleField 6-37
Singly linked list 8-29
Site licensing 1-45
Slider 4-32
SMACS 13-23
Sorting 14-45
Soundex 9-26
StackWindow 4-5
Stacks 8-6, 11-1, 11-15
Standard TSR interface 11-19
Static methods 2-4
StaticQueue 8-19
Status byte (for printing) 7-5
Stream 2-7, 8-90
Stream manager 8-94
Stream, memory 8-94
Stream, registration routines 6-50, 8-94
Strenczewilk, Don 9-5
String dictionary 8-67
String handling 1-2, 9-1
String management 8-67
String searching 4-198
StringArray 8-67
StringEditor 6-52, 6-116, 6-179
StringField 6-37
StringSet 8-67
StripPicture 6-14
Subfields 6-5, 8
Swap region 11-2
Swapping, TSR 11-2, 11-34
Swapping, disk/EMS 11-40
Swapping, exec manager 10-32
Swapping, messages 11-40
System requirements 1-4

T

Tab characters 5-20, 9-11
TaskView (see OmniView)
Technical support line 1-46
Terminate and Stay Resident 11-1
Text editor 1-2, 5-1, 5-3
Text editor, memo fields 5-37
Text file device drivers 7-8
TextAttr 3-6
TextField 6-15
ThwartSideKick 2-10
Time variables 6-206
TimeField 6-37
Toshiba P321SL or P341SL printers 7-58
TSR 11-1
TSR, unloading 11-20
Turbo Professional, converting from 15-16
TURBO3 3-3
TURBO5.MAC 13-23
Typecasting 8-31, 14-28

U

Unit dependencies 15-4
Unloading a TSR 11-20
UseAdjustableWindows 2-10
UseBCD 2-10
UseCrt 2-10
UseDates 2-10
UseHotSpots 2-10
UseMouse 2-10
UseScrollBars 2-10

UseShadows 2-10
User exit commands 3-95

V
Validation (see Fields)
VDISK.SYS 12-1
VGA 3-1, 3-7, 3-22, 3-35
Video attribute 3-1, 3-4, 3-48
Video memory 3-29
Video page 3-1, 3-24, 3-36, 3-39, 3-66, 3-79
Video segment 3-24, 4-173
Virtual methods 2-4
Virtual screen 4-173, 6-101

W
Warranty period 1-45
WFrame 4-21
White space 9-31
WindMax 3-6
WindMin 3-6
Window 4-8
Window, coordinates 3-33, 4-55
Window, headers 4-23
Window, hierarchy 4-1, 4-55, 4-138
Window, manager 4-56, 4-112
Window, mixing RawWindow and
    StackWindow 4-116
Window, moving and resizing 4-113
Window, objects 1-1, 4-1, 4-8
Window, saving 4-112
Window, scrolling 4-112
Window, separator bar 4-328
Window, shadows 4-21, 24
Window, sizing for PickLists 4-195
Window, stack 4-112, 15-15
Window, streams 4-57
Window-relative writing 4-103, 4-113
WINWOW 4-438
Wrap mode 6-62, 6-80
Word wrap 9-34
WordField 6-38

Y
YesNoField 6-38

Z
Zero-based arrays 8-158

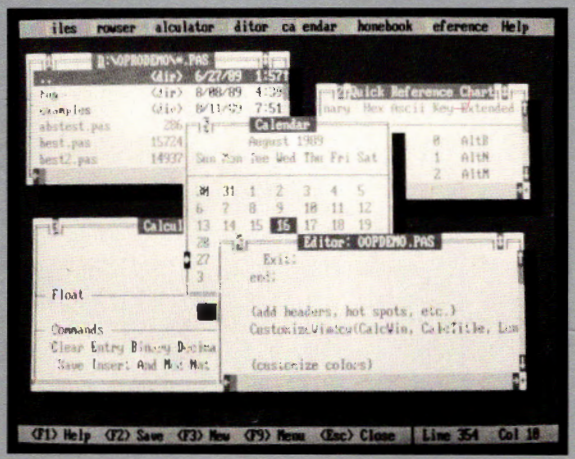
Object Professional™

A Powerful Library for Object-Oriented Programming in Pascal

Object-oriented programming multiplies your productivity by promoting reuse of proven software modules. Don't start from scratch with your objects! Use Object Professional 1.0, a huge library of over 100 powerful object types containing over 1000 methods.

Powerful User Interface Objects

The window objects let you use multiple overlapping and resizeable windows. Your windows will provide capabilities like:



- menus
- pick lists
- file browsing
- scrolling data entry screens with:
  - help capability
  - line editors
  - data validation and formatting
  - hidden and protected fields
  - matching capability for printed forms
- mouse support and scroll bars
- text editing

The window objects are completely flexible—you can create text-mode PM look-alikes or your own unique look and feel. Includes source generator for menus and entry screens.

Data Object Types . . . and System-Oriented Routines Too !

Build your programs from proven and documented object types like stacks, linked lists, virtual arrays, and more. System-oriented routines provide: ■ Swappable TSRs in only 6K of RAM ■ Interrupt management ■ EMS management ■ DOS and BIOS capabilities, and more.

Documentation: Clear and Complete

Object Professional includes clear, comprehensive documentation, on-line help, full source code, technical support, and hot demo programs. You pay no royalties when you use these routines.

Object Professional requires Turbo Pascal 6.0 or 5.5, PC-DOS or MS-DOS 2.0 or later, and an IBM PC, XT, AT, PS/2 or compatible.

TURBO Power

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

©TurboPower Software 1991